Investigating a Heterogeneous Data Integration Approach for Data Warehousing

Hao Fan

November 2005

A Dissertation Submitted to the University of London
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

School of Computer Science & Information Systems
Birkbeck College

To my parents, my wife and my son

Acknowledgments

I am especially grateful to my supervisor, Prof. Alexandra Poulovassilis, for her continued support and patient guidance.

My thanks are also due to the many colleagues at the School of Computer Science & Information Systems and London Knowledge Lab, especially Nigel Martin, Lucas Zamboulis, George Papamarkos, Dean Williams, Rachel Hamill for their precious friendship and timely help during my years at Birkbeck.

I thank all the AutoMed members at Birkbeck and Imperial College for helping to build a platform on which my research is based: Alex Poulovassilis, Peter McBrien, Michael Boyd, Sasivimol Kittivoravitkul, Nikolaos Rizopoulos, Nerissa Tong, Dean Williams, and Lucas Zamboulis.

Last, but not least, my thanks must go to my parents for their love, support and guidance; my wife, Fei, for being so supportive and for making my life meaningful; and my nine-month old son, Tianda, for bringing me so much happiness.

Abstract

Data warehouses integrate data from remote, heterogeneous, autonomous data sources into a materialised central database. The heterogeneity of these data sources has two aspects, data expressed in different data models, called *model heterogeneity*, and data expressed within different schemas of the same data model, called *schema heterogeneity*.

AutoMed[1] is an approach to heterogeneous data transformation and integration based on the use of reversible schema transformation sequences, which offers the capability to handle data integration across heterogenous data sources. So far, this approach has been used only for virtual data integration. In this thesis, we investigate the use of this approach for materialised data integration.

We investigate how AutoMed metadata can be used to express the schemas present in a data warehouse environment and to represent data warehouse processes such as data transformation, data cleansing, data integration, and data summarization. We discuss how the approach can be used for handling schema evolution in such a materialised data integration scenario. That is, if a data source or data warehouse schema evolves how the integrated metadata and data can also to be evolved so that the previous integration effort can be reused as much as possible. We then describe in detail how the approach can be used for two key data warehousing activities, namely data lineage tracing and incremental view

[1]See `http://www.doc.ic.ac.uk/automed/`

maintenance.

The contribution of this thesis is that we investigate for the first time how AutoMed can be used in a materialised data integration scenario. We show how the evolution of both data source and data warehouse schemas can be handled. We show how two key data warehousing activities, namely incremental view maintenance and data lineage tracing, are performed. This is also the first time that data lineage tracing and incremental view maintenance have been considered over sequences of schema transformations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Data Warehousing

A data warehouse consists of a set of materialised views defined over a number of data sources. It collects copies of data from remote, distributed, autonomous and heterogeneous data sources into a central repository to enable analysis and mining of the integrated information. Data warehousing and on-line analytical processing (OLAP) are essential elements of decision support, which has increasingly become a focus of the database industry. Many commercial products and services relating to data warehousing are currently available, and all of the principal data management system vendors, such as Oracle, IBM, Informix and MS SQL Server, have offerings in these areas.

Research problems in data warehousing include data warehouse architecture design, information quality and data cleansing, maintaining data warehouses, selecting views to materialise, Workflow data management [BCDS01], data lineage tracing in data warehouses, and so on. Comprehensive overviews of data warehousing and OLAP technology are given in [CD97, Wid95]. Currently, increasing numbers of data warehouses need to integrate data from a number of heterogeneous and autonomous data sources. Extending existing warehouse activities into heterogeneous database environments is a new challenge in data warehousing research.

The heterogeneity of these data sources has two aspects, data expressed in different data models, called *model heterogeneity*, and data expressed within different schemas of the same data model, called *schema heterogeneity*.

Up to now, most data integration approaches have been either *global-as-view* (*GAV*) or *local-as-view* (*LAV*) [Len02]. In GAV, the constructs of a global schema are described as views over local schemas[1]. In LAV, the constructs of a local schema are defined as views over a global schema. One disadvantage of GAV and LAV is that they do not readily support the evolution of both local and global schemas. In particular, GAV does not readily support the evolution of local schemas while LAV does not readily support the evolution of global schemas. Furthermore, both GAV and LAV assume one common data model for the data transformation and integration process, typically the relational data model.

Other approaches for managing distributed, heterogenous, and autonomous databases and database applications include *federated databases* [SL90, BIG94, SG97] and *middleware* [BCRP98, CEM01]. In contrast to data warehouses being materialised data integration scenarios, federated database systems are virtual data integration scenarios which use virtual federated schemas integrating schema information from distributed and autonomous source databases. They are an early example of the GAV approach. Global query processors are used to evaluate queries over federated schemas by accessing the data in the source

[1]A *view* in a database system is derived data defined in terms of stored data and/or possibly other views. View definitions are expressed as queries over their source data. A view can be materialised by storing the data of the view, and subsequent accesses of the materialised view can be much faster than recomputing it.

databases. The middleware approach presents a unified programming model to resolve heterogeneity, and facilitates communication and coordination of distributed components, so as to build systems that are distributed across a network [Emm00]. For undertaking data transformation or integration, middleware can adopt GAV, LAV or both approaches.

1.2 The BAV Data Integration Approach

AutoMed[2] supports a new data integration approach called *both-as-view* (*BAV*) which is based on the use of reversible sequences of primitive schema transformations [MP03a]. From these sequences, it is possible to derive a definition of a global schema as a view over the local schemas, and it is also possible to derive a definition of a local schema as a view over a global schema. BAV can therefore capture all the semantic information that is present in LAV and GAV derivation rules. A key advantage of BAV is that it readily supports the evolution of both local and global schemas, allowing transformation sequences and schemas to be incrementally modified as opposed to having to be regenerated.

Another advantage is that BAV can support data transformation and integration across multiple data models. This is because BAV supports a low-level data model called the HDM (hypergraph data model) in terms of which higher-level data models are defined. Primitive schema transformations add, delete or rename a single modelling construct with respect to a schema. Thus, intermediate schemas in a schema transformation/integration network can contain constructs defined in multiple modelling languages. Previous work has shown how relational, ER, OO, XML and flat-file data models can be defined in terms of the HDM [MP99a, MP99b, MP01].

[2]See `http://www.doc.ic.ac.uk/automed/`

AutoMed is an implementation of the BAV data integration approach. In previous work within the AutoMed project [PM98, MP99a], a general framework has been developed to support schema transformation and integration. So far, the BAV approach and AutoMed have been used only for virtual data integration. In this thesis, we investigate the use of the BAV approach for materialised data integration. We first investigate how AutoMed metadata can be used to express the schemas present in a data warehouse environment and to represent data warehouse processes such as data transformation, data cleansing, data integration, and data summarisation. We then discuss how schema evolution can be handled in such a materialised data integration scenario. That is, if a data source or data warehouse schema evolves how the existing warehouse metadata and data can also be evolved so that the previous integration effort can be reused. We then describe in detail how the approach can be used for two key data warehousing processes, namely data lineage tracing and incremental view maintenance.

1.3 Problem Statement

In order to use AutoMed for materialised data integration, there are four research problems considered in this thesis.

1. How AutoMed metadata can be used to express the schemas and processes such as data cleansing, transformation and integration in heterogeneous data warehouse environments, supporting both schema heterogeneity and model heterogeneity.

2. How AutoMed schema transformations can be used to express the evolution of a data source or data warehouse schema, either within the same

data model, or a change in its data model, or both; and how the existing warehouse metadata and data can also be evolved so that the previous transformation, integration and data materialisation effort can be reused.

3. How AutoMed metadata can be used for data lineage tracing in heterogeneous data warehouses, including what is the definition of data lineage in the context of AutoMed, and how the individual steps of AutoMed schema transformations can be used to trace data lineage in a step-wise fashion.

4. How AutoMed metadata can be used for incremental view maintenance in heterogeneous data warehouses. Here, we discuss how AutoMed can handle the problem of maintaining materialised data warehouse views if either the data or the schema of a data source change.

1.4 Dissertation Outline

The outline of this thesis is as follows:

Chapter 2 gives the background of this thesis, including a review of major issues in data warehousing.

Chapter 3 gives an overview of the AutoMed framework, at the level necessary for the work in this thesis, and discusses how AutoMed metadata can be used to express the schemas and processes of heterogeneous data warehousing environments.

Chapter 4 describes how AutoMed schema transformations can be used to express the evolution of schemas in a data warehouse. It then shows how to evolve the warehouse metadata and data so that the previous transformation, integration and data materialisation effort can be reused.

Chapter 5 develops a set of algorithms which use materialised AutoMed schema transformations for tracing data lineage. By materialised, we mean that all intermediate schema constructs created in the schema transformations are materialised, *i.e.* have an extent associated with them.

Chapter 6 generalises these algorithms to use arbitrary AutoMed schema transformations for tracing data lineage *i.e.* where intermediate schema constructs may or may not be materialised.

Chapter 7 discusses how AutoMed transformation pathways can be used for incrementally maintaining data warehouse views.

Finally, Chapter 8 gives our conclusions and directions of future work.

1.5 Dissertation Contributions

A formal approach has been chosen as the methodology of this research. We first investigate previous relevant work on data warehousing, schema evolution, data lineage tracing, and incremental view maintenance. We then investigate how the AutoMed data integration approach can be used for these activities in the context of heterogeneous data warehouse environments, develop new theoretical foundations and algorithms, and implement some of our algorithms.

The contribution of this thesis is that we investigate for the first time how the AutoMed heterogeneous data integration approach can be used in a materialised data integration scenario. We show how the evolution of both data source and data warehouse schemas can be handled. We show how two key data warehousing activities, namely incremental view maintenance and data lineage tracing, are performed. This is also the first time that data lineage tracing and incremental view maintenance have been considered over sequences of schema transformations.

Chapter 2

Overview of Major Issues in Data Warehousing

This chapter gives an overview of major issues in data warehousing. In Section 2.1, we discuss a definition of a data warehouse. Section 2.2 presents the architecture of a data warehouse system which includes the data sources, the staging area, the data warehouse itself and end-user applications and interfaces. Section 2.3 discusses a commonly-used data modelling technique in data warehousing, multidimensional data modelling. Section 2.4 discusses the processes of building, maintaining and using a data warehouse. Finally, Section 2.5 summarises the discussions of this chapter.

2.1 What is a Data Warehouse?

A data warehouse is a repository gathering data from a variety of data sources and providing integrated information for Decision Support Systems of an enterprise. In contrast to operational database systems which support day-to-day operations of an organisation and deal with real-time updates to the databases, data warehouses support queries requiring long-term, summarised information integrated from the data sources, and generally do not require the most up to date operational version of the data. Thus, updates to the primary data sources do not have to be propagated to the data warehouse immediately.

The definition of a data warehouse given in [Inm02] is:

> A data warehouse is a *subject-oriented*, *integrated*, *nonvolatile* and *time-variant* collection of data in support of management's decisions.

The first feature, *subject-oriented*, means that a data warehouse only includes the data that will be used for the organisation's Decision Support System (DSS) processes. In contrast, other database applications contain data for satisfying immediate functional or processing requirements, which may or may not have any use for decision support. The *subject* in the above definition denotes the aspect of the data used in DSS, such as the customers, products, services, prices and sales of the enterprise.

The second feature in the above definition is *integrated*. Data warehouses collect data from multiple data sources, which may be distributed, heterogeneous and autonomous. However, the warehouse data needs to be stored in a schema that satisfies the users' analysis requirements. Normally, source data is transformed and integrated before entering the data warehouse so that the focus of the warehouse users is on using the integrated data, rather than being concerned with the correctness or consistency of the source data.

The third feature in the above definition is *nonvolatile* which means that warehouse data are normally long-term, not updated in real-time and just refreshed periodically. In operational database systems, the data is normally the most up to date, and update operations such as inserting, deleting and changing data are frequently applied. In data warehouses, the data is used for DSS processes. Once the data is loaded into the data warehouse, the focus is on querying it, rather than inserting, deleting or changing it. However, a data warehouse also needs to be periodically refreshed in order to reflect updates in the primary data sources. Usually, alternate bulk storage is used to store the old data in the data warehouse. Purges of obsolete data are also carried out from time to time.

The last feature in the above definition is *time-variant*. Information from one past time point (the time the data warehouse was deployed) to the present may be contained in the data warehouse. Using this information, end users can analyse and forecast the progress and future trends of the enterprise. In contrast, operational database applications mainly consider only current data.

In summary, a data warehouse is built for DSS analysts or managers in an enterprise, who may be non-technical users, to easily access in their business context the widespread information across the enterprise. It is a single, complete, consistent accumulation of data obtained from a variety of sources which may be remote, distributed, heterogeneous and autonomous. In order to take advantage of this data, the basic functionalities of a data warehouse are gathering, cleansing, filtering, transforming, integrating and reorganising the source data into a repository with a single schema which satisfies the users' analysis requirements. Thus, data warehousing is not a static solution but an evolving process.

2.2 Data Warehouse Architecture

A data warehouse system consists of several components: the data sources, the staging area, the data warehouse itself and end-user applications and interfaces, as illustrated in Figure 2.1. Brief descriptions of each component are given below.

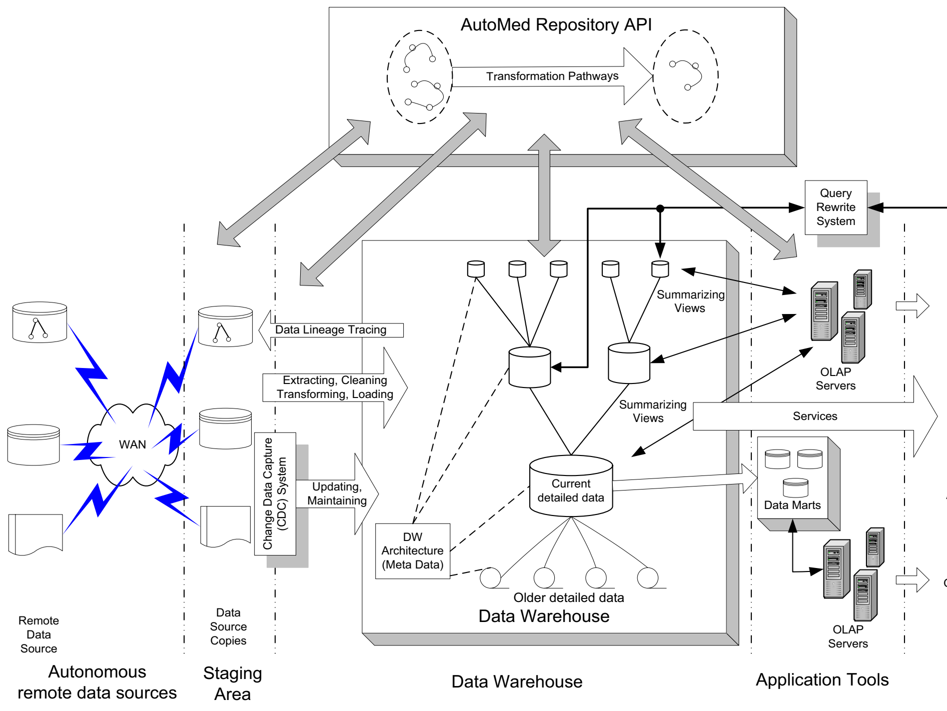


Figure 2.1: Basic Components of a Data Warehouse System Using AutoMed

Data Sources The data sources provide the original data of the data warehouse. A data warehouse may integrate data from multiple autonomous and heterogeneous data sources, which could be either remote or local, and not under the control of the data warehouse users and administrators. In addition, the data sources may be structured (e.g., relational databases), semi-structured (e.g., XML or RDF files) or flat files. Such arbitrary data sources pose several challenges to warehouse builder: to create a uniform repository integrating these data; to design easily understandable data warehouse schemas; and to express the transformations between the data source and data warehouse schemas.

Staging Area The staging area keeps whole copies of the data sources and brings them under the control of the data warehouse administrator. The data in the staging area may be heterogeneous and contain "dirty" (*e.g.* duplicate or inconsistent) data. No end-user query services are available in this area, that is, the warehouse users cannot access the data in the staging area.

Data Warehouse The data warehouse contains the integrated data used to support the DSS processes. In contrast to the staging area, data in the data warehouse itself have a uniform schema and have been cleansed by removing dirty data. The processes of data cleansing and data transformation happen before loading data into the data warehouse.

The data warehouse typically consists of following components:

- Detailed Data: The detailed data is the lowest level of source information necessary for supporting the DSS processes. It is normally stored in a single repository such as a relational or object-oriented database. The detailed data includes current detailed data and older detailed data. From the staging area to the detail data, the data needs to be transformed, cleansed, loaded and integrated. These processes compose a major part of building a data warehouse.

- Summarised Data: The summarised data is derived from the detailed data, in order to allow faster processing of specific DSS functionality. For example, suppose the detailed data contains a relational table Sales(<u>ProductID</u>, <u>LocationID</u>,<u>TimeID</u>,SalesAmount). The summarised data may contain tables ProductSalesByLocation(<u>ProductID</u>,<u>LocationID</u>, SalesAmount) summarising the total sales for products at locations; ProductSalesByTime(<u>ProductID</u>, <u>TimeID</u>, SalesAmount) summarising the total sales for products over time periods; LocationSalesByTime(<u>LocationID</u>, <u>TimeID</u>, SalesAmount) summarising the total sales for locations over time periods; TotalProductSales(<u>ProductID</u>, SalesAmount) summarising the total sales for products; TotalLocationSales

(LocationID, SalesAmount) summarising the total sales for locations; TotalTimeSales(ProductID, SalesAmount) summarising the total sales over time periods.

The summarised data are defined as *views* over the detailed data or over other summarising views. Views in the data warehouse can be virtual or materialised. How to maintain these views, especially materialised ones, has been one of the key issues of data warehousing research [GM99, Don99].

- Metadata: A data warehouse not only provides integrated data, but also provides information about the content and context of the data, *i.e. metadata*. This metadata provides a directory of the structure of the warehouse contents. It provides information about the warehouse schema, and also about the mappings between the data in the data warehouse, such as from the data sources to the detailed data and from the detailed data to the summarised data. In Figure 2.1, we show the metadata being stored in an AutoMed repository, where it can be accessed by the data warehouse users and administrators.

End-User Applications and Interfaces The end-user applications and interfaces provide a way for warehouse users to access warehouse data. In particular, *data marts* can be created over the data warehouse for different categories of DSS users. Data marts are defined from the warehouse data for specific DSS requirements of the enterprise. In contrast to the summarised data, data marts can have different data models and schemas from the ones of the detailed data of the warehouse. In practice, the same tools used to load the data warehouse database can be used to load the data marts, for example Oracle Warehouse Builder[1], IBM

[1]See `http://www.oracle.com/technology/documentation/warehouse.html`

Data Warehouse Manager[2], or Microsoft Data Transformation Services[3].

The problem of *query rewriting*, also known as *answering queries using views*, has also received much attention in database research [Lev00]. Query rewriting aims to find efficient methods of answering a query using a set of previously materialised views over the database tables, rather than accessing the base tables themselves. In data warehousing, it is relevant to problems such as query optimisation, materialised view maintenance and data warehouse design. We do not address the problem of query rewriting in this thesis, but it may be an important area of future research in the AutoMed project.

2.3 Data Warehouse Modelling

Data warehouse modelling is the process of designing the schemas of the detailed and summarised data of the data warehouse. The aim of data warehouse modelling is to design a schema representing the reality, or at least a part of the reality, which the data warehouse is required to support.

Data warehouse modelling is an important stage of building a data warehouse for two main reasons. Firstly, through the schema, data warehouse users have the ability to visualise the relationships among the warehouse data, so as to use them with greater ease. Secondly, a well-designed schema allows an effective data warehouse architecture to emerge, to help reduce the cost of implementing the warehouse and improve the efficiency of using it.

Data modelling in data warehouses is rather different from data modelling in operational database systems. The main functionality of data warehouses is to support DSS processes. Thus, the aim of data warehouse modelling is to make the data warehouse efficiently support complex queries on long-term information.

[2]See `http://www-306.ibm.com/software/data/db2/datawarehouse/`

[3]See `http://www.microsoft.com/sql/evaluation/features/datatran.asp`

In contrast, data modelling in operational database systems focusses on efficiently supporting simple transactions in the database such as retrieving, inserting, deleting and changing data. Moreover, data warehouses are designed for users with general information knowledge about the enterprise, whereas operational database systems are more oriented toward use by software specialists for creating specific applications.

Modelling warehouse data requires information about both the source data and the target warehouse data. The source data can be treated as inputs which are transformed into the target warehouse data. How this transformation happens is required to be reflected in data warehouse modelling.

Multidimensional data modelling is a commonly-used technique to conceptualise and visualise schemas by using the major components of the business, such as customers, products, services, prices and sales. This data modelling technique is especially used for summarising and rearranging data and presenting views of the data to support DSS. Particularly, multidimensional data modelling focuses on numeric data such as sales, counts, balances and costs.

In multidimensional data modelling, the data warehouse is designed to collect $facts$ on one or more $measures$, each measure depending on a set of $dimensions$. For example, a sales measure may depend on three dimensions: products, times and locations.

Facts are collections of related data items, which are stored within *fact tables* in the data warehouse. *Dimensions* are collections of the items of one component of the business, such as the products dimension, the times dimension and the locations dimension for sales. The items of a dimension are stored within a *dimension table* in the data warehouse.

The primary key of a fact table is a concatenation of the primary keys of one or more dimension tables. Thus, every row in the fact table is associated with one and only one row from each dimension table.

Measures are the non-key attributes of fact tables, and they represent information relating to the dimensions key attributes of the fact table.

The non-key attributes of a dimension table may be organised as a *dimension hierarchy*. For example, the times dimension may consist of the dates, months and weeks attributes; the products dimension may consist of the category, model and producer attributes; and the locations dimension may consist of the city, region and country attributes.

There are two kinds of schemas used in multidimensional data modelling: *star schemas* and *snowflake schemas*. A star schema typically has one fact table, and a set of smaller tables. Figure 2.2 (Left) below gives an example of a star schema. The links between the primary keys of the fact table and the foreign keys in the dimension tables can be visualised as a radial pattern with the fact table in the middle.

The dimension tables may contain data redundancies. For example, in the dimension table Locations(LocationID,Address,City,Region,Country), the City and Region information may be repeatedly stored for the locations in the same cities. This kind of data redundancy incurs storage overheads and may lead to update anomalies and poor update performance.

If necessary, snowflake schemas can be used to avoid such data redundancies. A snowflake schema is the result of normalizing the dimensions of a star schema, in which there are links between primary keys and foreign keys of tables in the dimension hierarchy. Figure 2.2 (Right) is an example of a snowflake schema.

However, fully normalizing the dimension tables may not be necessary in a data warehouse environment. Since there are generally no updates occurring to individual rows in the dimension tables, although new rows may be added when the data warehouse is refreshed with new data, and existing rows may be deleted

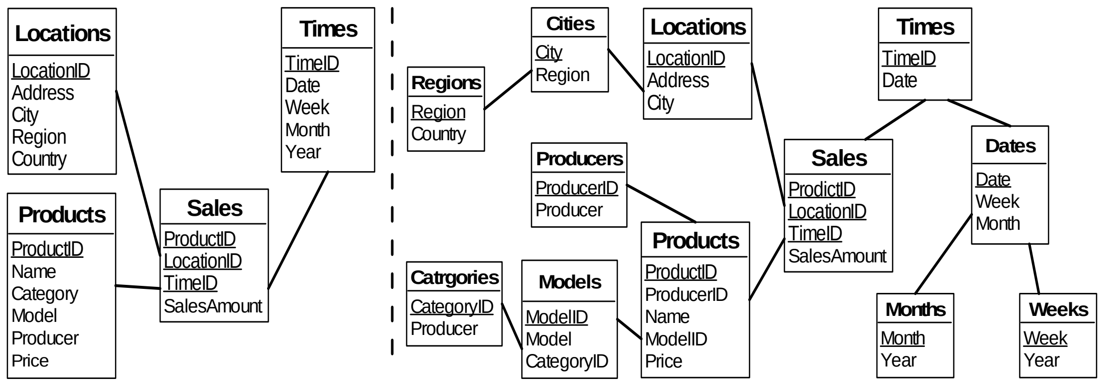


Figure 2.2: (Left) Star Schema (Right) Snowflake Schema

when the data warehouse is purged of out-of-date data, the issue of update anomalies and poor update performance will generally not arise in the data warehouse. In addition, the storage consumption of the data warehouse is dominated by the fact tables and the space saved by normalizing the dimension tables would generally be comparatively small. Moreover, un-normalized dimension tables can reduce the time required to combine information in the fact table with dimension information, which is a main performance criterion of a data warehouse.

2.4 Data Warehouse Processes

The objective of supporting DSS queries over a data warehouse requires a set of data warehouse processes that are far more complex than just collecting data from the remote data sources and then querying them. In this section, we discuss the processes of *building*, *maintaining* and *using* the data warehouse. In particular, building the data warehouse includes *extracting*, *cleansing*, *transforming*, *loading*, *summarising* data and creating *data marts*; maintaining the data warehouse is the process of refreshing materialised views in the warehouse and the data marts; and using the data warehouse includes developing and using the end-user applications, as well as special functionalities of the data warehouse, such as data lineage tracing.

2.4.1 Data Extraction

Data are extracted from the data sources into the staging area for integration into the data warehouse. This is the first step of building the data warehouse. Data extraction does not involve complex algebraic database operations such as `join` and aggregate functions. It focuses on determining which remote data is required to be extracted, and bringing the data into the staging area. The data sources may be very complex and poorly documented, so that data extraction design and performance are often the time-consuming tasks in the building process [Lan02].

Data have to be extracted not only once, but several times in a periodic manner to supply the changes to the data warehouse and keep it up-to-date. Thus, data extraction is not only used in building the data warehouse, but also used in maintaining the data warehouse.

There are two kinds of strategies of data extraction: *full extraction*, where the entire files or tables of the data sources are extracted to the staging area; and *incremental extraction*, only the data that has been changed since a well-defined event back in history will be extracted at a specific point of time. The event may be the last time of successful extraction or a more complex business event like the last sale day of a fiscal period [Lan02].

Full extraction reflects all data currently available in the data sources, and there is no need to keep track of the changes to a data source since the last successful extraction. The source data will be provided as a whole and no additional information, such as time-stamps, is necessary regarding the source site.

Incremental extraction can make the data extraction process much more efficient, and is especially useful when incremental view maintenance (see Section

2.4.5 below) has been selected as the maintenance strategy. However, for many data sources, identifying the recently modified data may be difficult or intrusive to the operations of the data sources, which is beyond the control of the data warehouse builder.

Normally, the data sources cannot be modified by the data warehouse builder, nor can their performance or availability be affected by the data extraction process. Because of the independence of the data sources, data warehouses normally do not use incremental extraction as the strategy for data extraction and instead use full extraction. After full extraction, the entire extracted data from the data sources can be compared with the previous extracted data to identify the changed data, so that delta changes can be captured for maintaining the warehouse data (this happens in the staging area). This approach may not have significant impact on the data sources, but it clearly can place a considerable burden on the data warehouse processes, particularly if the data volumes are large.

Data extraction incorporates the processes of data *transportation* and data *loading*, which move data from one data system to another. The most common requirements of data transportation are moving data from the data sources to the staging area, from the staging area to the data warehouse, and from the data warehouse to the data marts. Data loading is data transportation specifically relating to loading the detailed data into the data warehouse.

In practice, Load is a command in many commercial database systems. For example, the Oracle SQL*Loader utility is used to move data from flat files into Oracle tables, which is faster than using a series of SQL INSERT statements because no locking or logging takes place. Similarly, Transact-SQL and the bcp utility from Microsoft[4] can be used to load data into SQL Server databases. There

[4]See `http://msdn.microsoft.com/library/`.

are also available many commercial tools for data extraction and loading in data warehouses, such as Oracle Warehouse Builder[5], IBM Data Warehouse Manager[6], and Microsoft Data Transformation Services[7].

2.4.2 Data Transformation

The data sources of a data warehouse may conform to multiple schemas, while the data warehouse has a single schema. Heterogeneous source data have to be transformed into the data warehouse schema before loading into the data warehouse.

Two kinds of data transformations are often used in data warehousing: *multistage data transformations* and *pipelined data transformations* [Lan02]. Multistage transformations implement each different transformation as a separate operation and create separate, temporary staging tables to store incremental results of each step. This is a common strategy and makes the transformation process easily monitored and restarted. However, a disadvantage of multistage data transformations is high space and time costs.

With pipelined data transformations, there are no temporary staging tables. Instead data is transformed as it is loaded into the data warehouse. This consequently increases the difficulty of monitoring and may require some similarity between the source data and the target data, *e.g.* both of them have schemas specified within the same data model. For example, the commercial data management tool PgManager[8] can be used to transform data in Excel tables, Access databases or TXT files and load them into PostgreSQL databases.

[5]See `http://www.oracle.com/technology/documentation/warehouse.html`
[6]See `http://www-306.ibm.com/software/data/db2/datawarehouse/`
[7]See `http://www.microsoft.com/sql/evaluation/features/datatran.asp`
[8]See `http://sqlmanager.net/products/postgresql/manager/`

2.4.3 Data Cleansing

Extracting data from remote data sources, especially from heterogeneous data sources, can bring erroneous and inconsistent information into the data warehouse. Data warehouses usually face this problem, in their role as repositories for information derived from multiple data sources within and across enterprises. Thus, before loading data from the staging area to the data warehouse, *data cleansing* is normally required [RD00]. Data cleansing is a process which deals with detecting and removing errors and inconsistencies from the source data in order to improve the data quality of the data warehouse.

The problems of data cleansing include *single-source* problems and *multi-source* problems [RD00]. Single-source cleansing cleans dirty data from one data source. This process involves formatting and standardizing the source data, such as adding a key to every source record and decomposing some dimensions into sub-dimensions according the requirement of the warehouse, *e.g.*, decomposing an Address dimension into LocationID, Number, Street, City, Zip and Country dimensions. Multi-source cleansing considers several data sources when undertaking the cleansing process. Multi-source cleansing may include merging data from multiple data sources.

Figure 2.3 illustrates an example of merging data from multiple data sources. The Customer and Client databases are integrated into the Customers database. Records existing in one data source, Customer or Client, remain in the Customers database under the transformed schema. As to records existing in both data sources, information from the more reliable source can be transformed into the target database.

For each of these two data cleansing problems, there are two possible scenarios: *schema-level* and *instance-level* [RD00]. Schema-level problems can be addressed by evolving the schema(s) as necessary. Instance-level problems, on the other

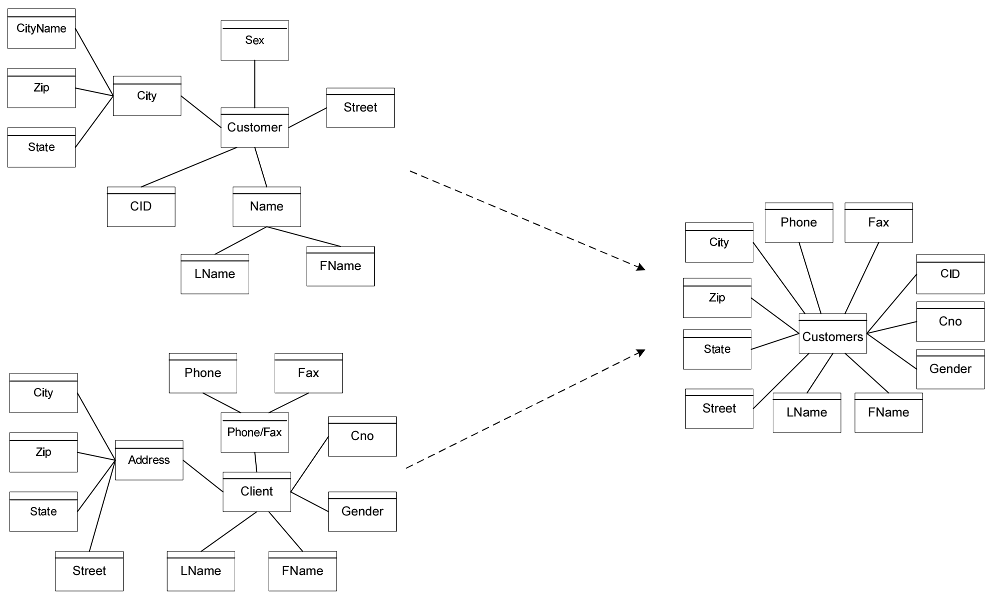


Figure 2.3: Merging Data from Multiple Data Sources

hand, refer to errors and inconsistencies in the actual data contents which are not visible at the schema level. Below, we discuss data cleansing problems for both single and multiple data sources, and for both schema-level and instance-level problems:

Single-Source Cleansing Single-source, schema-level problems arise when the source data model violates the schema used for the data warehouse. For example, the source data may be XML files while the schema used for the data warehouse is relational, or relational databases with different schemas are used to represent the same information in a data source and in the data warehouse.

Single-source, instance-level problems include *value*, *attribute* and *record* problems. Value problems occur within a single value and include problems such as a missing value, a mis-spelled value, a mis-fielded value (*e.g.* putting a city name in a country attribute), embedded values (putting multiple values into one attribute value), using an abbreviation or a mis-expressed value (*e.g.* using the wrong order of first name and family name within a name attribute).

Attribute problems relate to multiple attributes in one record and include problems such as dependence violation (*e.g.* between city and zip, or between birth-date and age).

Record problems relate to multiple records in the data source, and include problems such as duplicate records or contradictory records.

Multi-Source Cleansing Multi-source, schema-level problems include *attribute* and *structure* conflicts. Attribute conflicts arise when different sources use the same name for different constructs (homonyms) or different names for the same construct (synonyms). Structure conflicts arise when the same information is modeled in different ways in different schemas. For example, information about customers may be stored in relational databases and XML documents, or in relational databases with different schemas (*e.g.* regionCustomer(name,location,service) storing customer information according to their location and services they use; and wholeSaleCustomer(name,address) and retailCustomer(name,address) storing customer information in different tables according to their service type.).

Multi-source, instance-level problems include *attribute*, *record*, *reference* and *data source* problems. Attribute problems include different representations of the same attribute in different schemas (*e.g.* `Yes/No` vs `True/Fasle` in a maritalStatus attribute) or a different interpretations of the values of an attribute in different schemas (*e.g.* US Dollar vs Euro in a currency attribute).

Record problems include duplicate records or contradictory records among different data sources.

Reference problems occur when a referenced value does not exist in the target schema construct and can be resolved by replacing the dangling references by `Null` values.

Data source problems relate to whole data sources, for example, aggregation at different levels of detail in different data sources (*e.g.* sales may be recorded per product in one data source and per product category in another data source).

In Chapter 3 below, we will discuss how AutoMed schema transformations can be used to express the process of data cleansing, both single- and multi-source. A large number of commercial tools of varying functionalities are available to support data cleansing[9]. These normally focus on specific data cleansing problems, such as address correction (*e.g.* QuickAddress Batch[10] and AddressAbility™[11]) and removal of duplicates (*e.g.* DoubleTake[12]). In the research arena, examples include the ARKTOS tool for data cleansing and transformation by Vassiliadis *et al.* [VVSK00], the IntelliClean tool for knowledge-based intelligent data cleansing by Lee and Low *et al.* [LLL00, LLL01], the interactive data cleansing system Potter's Wheel by Raman *et al.* [RH01], and the extensible data cleansing tool AJAX by Galhardas *et al.* [GFSS00, GFS$^+$01a]. All of these research tools consider the problem of data cleansing more generally than the commercial tools.

2.4.4 Data Summarisation

Data summarisation is the process of creating the summarising data in the data warehouse. As discussed before, the summarising data are views over the detailed data and possibly other views, and they may or may not be materialised. The main usage of materialised views is to increase the speed of queries over the warehouse data and also to allow query rewriting.

However, a problem relating to materialised views is *view maintenance.* If the

[9]See `http://web.tagus.ist.utl.pt/ helena.galhardas/cleaning.html` for a list of commercial data cleansing tools.
[10]See `http://www.qas.com/address-correction-software.asp`
[11]See `http://www.inforouteinc.com/prodA-1.html`
[12]See `http://www.tech4t.co.uk/doubletake/`

detailed data in the data warehouse is updated, the materialised views have to be refreshed also so as to keep them up-to-date [GM99, Don99].

2.4.5 Data Warehouse Maintenance

The issue of view maintenance in data warehouses has been widely discussed in the literature [GM99, Don99, CW91, GMS93, CGL$^+$96, Qua96, PSCP02, ZGMHW95, ZGMW98, AASY97], and many view maintenance policies and algorithms have been developed. Logically, there are two kinds of view maintenance approaches, *fully recomputing* and *incrementally refreshing*; while temporally, three kinds of view maintenance approaches may be adopted, *periodic maintenance*, *on-commit maintenance* and *on-demand maintenance* [GM99].

Fully recomputing means that if a data source is updated, the view will be refreshed by recomputing it from scratch. On the other hand, incrementally refreshing computes the changes to the view rather than recomputing all the view data. Incrementally refreshing a view can be significantly cheaper than fully recomputing the view, especially if the size of the materialised view is large compared to the size of the change.

A periodically maintained view is called a *snapshot*, and is generally used for integrating data from remote data sources, such as from the Internet. A snapshot has a lower consistency level between the view and the data sources than on-commit maintenance, but is easy to implement.

On-commit view maintenance is also referred to as *immediate view maintenance* [GM99], which means that views are refreshed every time an update transaction commits. Using an immediate view maintenance strategy, we can ensure that the materialised views will always contain the latest committed data. However, it increases the time overhead of committing update transactions.

The on-demand view maintenance policy can control the time that view maintenance occurs — materialised views are refreshed when a refresh command is explicitly issued. One kind of on-demand view maintenance is *on-queried view maintenance*, which means the maintenance procedure is performed only when the view is used or queried. This may reduce the overhead of the view maintenance process in a data warehouse if some views are seldom used [Eng02].

Both the periodical and on-demand view maintenance policies are a kind of *deferred view maintenance* strategy [GM99, CGL$^+$96]. Both policies use the post-update data sources and their changes to maintain the views. In contrast, the on-commit (immediate) view maintenance policy uses the pre-update data sources and the changes to them to maintain the views. One disadvantage of immediate view maintenance is that each update transaction incurs the overhead of refreshing the views, and this overhead increases with the number of views and their complexity.

In data warehousing environments, immediate view maintenance is generally not possible, since administrators of data sources may not know what views exist in the data warehouse, and data warehouse administrators may not be able to access the changes to the data sources directly. Deferred view maintenance can be performed periodically, or on-demand when certain conditions arise, and is generally used as the view refreshment policy in data warehousing environments.

Combining the maintenance logic and maintenance time, there are therefore six possible view maintenance strategies: immediate incremental, immediate recompute, periodic incremental, periodic recompute, deferred incremental and deferred recompute maintenance [Eng02, ECL03].

The view maintenance approach discussed by Gupta and Quass *et al.* in [GJM96, QGMW96] is to make views *self-maintainable*, which means that materialised views can be refreshed by only using the content of the views and the updates to the data sources, and not requiring to access the data in any data source. References [Huy97], [VM97] and [LLWO99] also discuss view maintenance problems pertaining to self-maintenance for views in data warehousing environments, focusing on select-projection-join (SPJ) views. Such a view maintenance approach usually needs auxiliary materialised views to store additional information. Whether these auxiliary materialised views are also self-maintainable, with the original views acting as the auxiliary data, is important to this research issue. We are not considering self-maintainability of views in this thesis.

Materialised warehouse views need to be maintained either when the data of a data source changes, or if there is an evolution of a data source schema. In Chapter 4 of this thesis we discuss how AutoMed transformation pathways can be used to express schema evolutions in a data warehouse. In Chapter 7 of this thesis we discuss incrementally refreshing materialised warehouse views when the data of a data source changes.

2.4.6 Data Lineage Tracing

Sometimes what is needed is not only to analyse the data in a data warehouse, but also to investigate how certain warehouse information was derived from the data sources. Given a data item t in the data warehouse, finding the set of source data items from which t was derived is termed the *data lineage tracing* problem [CWW00]. Supporting data lineage tracing in data warehousing environments has a number of applications: in-depth data analysis, on-line analytical mining (OLAM), scientific databases, authorization management, and schema evolution of materialised views [BB99, WS97, CWW00, GFS$^+$01b, FJS97].

In Chapter 3 of this thesis we discuss how AutoMed schema transformation pathways can be used to express the main processes of heterogeneous data warehousing environments, including data transformation, cleansing, integration, summarisation and creating data marts. In Chapters 5 and 6 we then address the issues of data lineage tracing over AutoMed schema transformation pathways, including: the definitions of data lineage in the context of AutoMed; the problem of derivation ambiguity in data lineage tracing; formulae for data lineage tracing based on a single transformation step; algorithms for data lineage tracing along a sequence of transformation steps; and handling virtual transformation steps, *i.e.* steps whose results are not materialised.

2.5 Discussion

This chapter has given an overview of the major issues in data warehousing. We first introduced the definition of a data warehouse, and indicated that data warehouses integrate data from distributed, autonomous, heterogeneous data sources in order to support the DSS processes of an enterprise. The basic components of a data warehouse system include the data sources, the staging area, the data warehouse itself and the end-user applications and interfaces. We discussed multidimensional data modelling. The data warehouse processes described in this chapter were: building a data warehouse, including data extraction, data transformation, data cleansing, data loading and data summarisation; maintaining a data warehouse; and data lineage tracing.

In the rest of this thesis, we will discuss how AutoMed metadata can be used to represent the data models and schemas of a data warehouse and the semantic relationships between them. We will also develop a set of algorithms which use AutoMed transformation pathways for incremental view maintenance and data lineage tracing in the data warehouse. Our algorithms consider in turn each transformation step in a transformation pathway in order to apply incremental view maintenance and data lineage tracing in a stepwise fashion. Thus, our algorithms are useful not only in data warehousing environments, but also in any data transformation and integration framework based on sequences of schema transformations, such as peer-to-peer and semi-structured data integration environments.

Chapter 3

Using AutoMed Metadata for Data Warehousing

3.1 Motivation

In data warehouse environments, metadata is essential since it enables activities such as data transformation, data integration, view maintenance, OLAP and data mining. Due to the increasing complexity of data warehouses, metadata management has received increasing research focus recently [MSR99, HMT00, BTM01, CB02].

Typically, the metadata in a data warehouse includes information about both the data and the data processing. Information about the data includes the schemas of the data sources, warehouse and data marts, ownership of the data, and time information such as the time when the data was created or last updated. Information about the data processing includes rules for data extraction, cleansing and transformation, data refresh and data purging policies, and the lineage of migrated and transformed data.

Up to now, in order to transform and integrate data from heterogeneous data

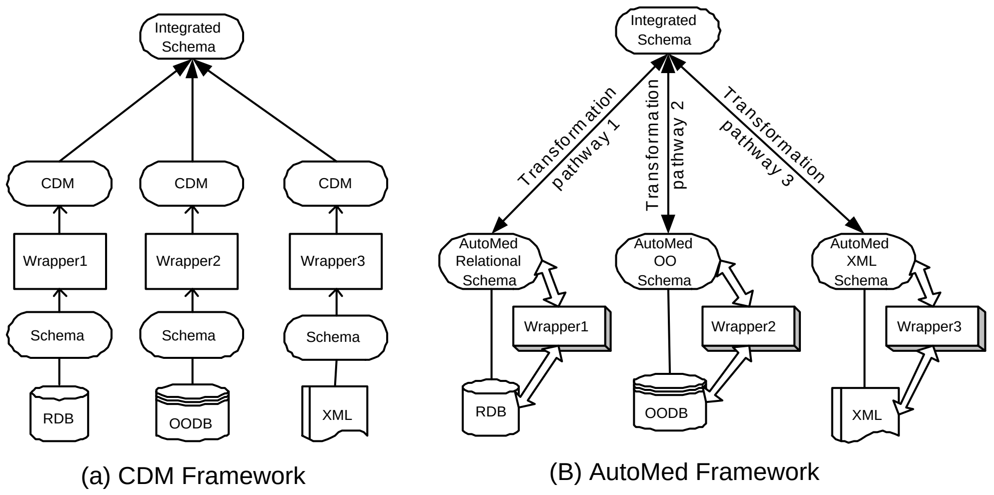


Figure 3.1: Frameworks of Data Integration

sources, a conceptual data model (CDM) has been used as the *common data model*, *i.e.* as the data model to which the detailed and summarised data of the data warehouse conform, and into which source data are translated. This approach assumes a single CDM for the data transformation and integration process — see Figure 3.1(*a*). Each data source[1] has a wrapper for translating its schema and data into the CDM of the detailed data. The schema of the summarised data is then derived from these CDM schemas by means of view definitions, and is expressed in the same modelling language as them.

For example, [HA01] uses the relational data model as the CDM; [MK00, CD97, TKS01] use a multidimensional model; [GR98] describes a framework for data warehouse design based on its Dimensional Fact Model; [CGL$^+$99, Bek99, TBC99, HLV00] use an ER model or extensions of it; and [VSS02] presents its own conceptual model and a set of abstract transformations for data extraction-transformation-loading (ETL).

This traditional CDM framework has a number of drawbacks. Firstly, since

[1]For the rest of the thesis, by *data source* we mean the copy of the remote data that has been brought into the staging area (unless otherwise indicated).

they are both high-level conceptual data models, semantic mismatches may exist between the CDM and a source data model, and there may be a loss of information between them. Secondly, if a source schema changes, it is not straightforward to evolve the view definitions of the integrated schema constructs in terms of source schema constructs. Finally, the data transformation and integration metadata is tightly coupled with the CDM of the particular data warehouse. If the warehouse is to be redeployed on a platform with a different CDM, it is not easy to reuse the previous warehouse implementation.

AutoMed is an implementation of the BAV data integration approach which adopts a low-level hypergraph-based data model (HDM) as its common data model for heterogeneous data transformation and integration[2]. So far, research has focused on using AutoMed for virtual data integration. This chapter describes how AutoMed can also be used for materialised data integration, in particular for expressing the data transformation and integration metadata, and using this metadata to support warehouse processes such as data cleansing, populating the warehouse, incrementally maintaining the warehouse data after data source updates, and tracing the lineage of warehouse data.

Using AutoMed for materialised data integration, the data source wrappers translate the source schemas into their equivalent specification in terms of AutoMed's low-level HDM — see Figure 3.1(*b*). AutoMed's schema transformation facilities can then be used to incrementally transform and integrate the source schemas into an integrated schema. The integrated schema can be defined in any modelling language which has been specified in terms of AutoMed's HDM. We will examine in this chapter the benefits of this alternative approach to data transformation/integration in data warehousing environments.

[2]See `http://www.doc.ic.ac.uk/automed` for a full list of technical reports and papers relating to AutoMed.

In the rest of this chapter, Section 3.2 gives an overview of the AutoMed framework to the level of detail necessary for this thesis. This includes a discussion of the HDM data model, the query language supported by AutoMed, the AutoMed transformation pathways and the AutoMed Repository API. Section 3.3 shows how AutoMed metadata has enough expressiveness to describe the data integration and transformation processes in a data warehouse, including expressing data transformation, data cleansing, data integration, data summarisation and creating data marts. Section 3.4 discusses how the AutoMed metadata can be used for some key data warehousing processes, including populating the data warehouse, incrementally maintaining the warehouse data, and tracing the lineage of the warehouse data. Section 3.5 discusses the benefits of our approach.

An earlier paper [The02] proposed using the HDM as the common data model for both virtual and materialised integration, and a hypergraph-based query language for defining views of derived constructs in terms of source constructs. However, that paper did not focus on expressing data warehouse metadata, or on warehouse processes such as data cleansing or populating and maintaining the warehouse.

3.2 The AutoMed Framework

3.2.1 HDM Data Model

The basis of AutoMed data integration system is the low-level **hypergraph data model (HDM)** [PM98, MP99b]. Facilities are provided for defining higher-level modelling languages in terms of this lower-level HDM. An HDM schema consists of a set of nodes, edges and constraints, and so each modelling construct of a higher-level modelling language is specified as some combination of HDM nodes, edges and constraints.

One advantage of using a low-level common data model such as the HDM is that semantic mismatches between high-level modelling constructs are avoided. Another advantage is that the HDM provides a unifying semantics for higher-level modelling constructs and hence a basis for automatically or semi-automatically generating the semantic links between them — this is ongoing work being undertaken by other members of the AutoMed project (see for example [ZP04, Riz04]).

A **schema** in the HDM is a triple $\langle Nodes, Edges, Constraints \rangle$. A **query** over a schema is an expression whose variables are members of $Nodes \cup Edges$. In this framework, the query language is not constrained to a particular one. However, the AutoMed toolkit supports a functional query language as its **intermediate query language (IQL)** — see Section 3.2.2 below.

Nodes and *Edges* define a labeled, directed, nested hypergraph. It is nested in the sense that edges can link any number of both nodes and other edges. It is a directed hypergraph because edges link sequences of nodes or edges. *Constraints* is a set of boolean-valued queries over the schema which are satisfied by all instances of the schema. In AutoMed, constraints are expressed as IQL queries. Nodes are uniquely identified by their names. Edges and constraints have an optional name associated with them.

The constructs of any higher-level modelling language $\mathcal{M}$ are classified as either **extensional constructs** or **constraint constructs**, or both. Extensional constructs represent sets of data values from some domain. Each such construct in $\mathcal{M}$ is represented using a configuration of the extensional constructs of the HDM *i.e.* of nodes and edges. There are three kinds of extensional constructs:

- **nodal** constructs may exist independently of any other constructs in a model. Such constructs are identified by a **scheme** consisting of the name of the HDM node used to represent that construct. For example, in the ER model, entities are nodal constructs since they may exist independently of each other. An ER entity `e` is identified by a scheme $\langle\langle \mathsf{e} \rangle\rangle$.

- **link** constructs associate other constructs with each other and can only exist when these other constructs exist. The extent of a link construct is a subset of the cartesian product of the extents of the constructs it depends on. A link construct is represented by an HDM edge. It is identified by a **scheme** that includes the name (and/or other identifying information) of constructs it depends on. For example, in the ER model, relationships are link constructs since they associate other entities. An ER relationship `r` between two entities `e1` and `e2` is identified by a scheme $\langle\langle \mathsf{r}, \mathsf{e1}, \mathsf{e2} \rangle\rangle$.

- **link-nodal** constructs are nodal constructs that can only exist when certain other constructs exist, and that are linked to these constructs. A link-nodal construct has associated values, but may only exist when associated with other constructs. It is represented by a combination of an HDM node and an HDM edge and is identified by a **scheme** including the name (and/or other identifying information) of this node and edge. For example, in the ER model, attributes are link-nodal constructs since they have an extent and must always be linked to an entity. An ER attribute `a` of an entity `e` is identified by a scheme $\langle\langle \mathsf{e}, \mathsf{a} \rangle\rangle$.

Finally, a **constraint** construct has no associated extent but represents restrictions on the extents of the other kinds of constructs. It limits the extent of the constructs it relates to. For example, in the ER model, generalisation hierarchies are constraints since they have no extent but restrict the extent of each subclass entity to be a subset of the extent of the superclass entity; similarly, ER relationships and attributes have cardinality constraints.

Previous work has shown how relational, ER, OO [MP99b], XML [MP01, Zam04] and flat-file [BKL+04] modelling languages can be defined in terms of the HDM. After a modelling language has been defined in terms of the HDM (via the API of AutoMed's Model Definition Repository — see Section 3.2.4 below), a set of primitive transformations is automatically available for the transformation of schemas defined in the language. Section 3.2.3 below will discuss AutoMed transformations.

In this section, we next illustrate how a simple relational model, simple XML data model and simple multidimensional data model can be represented in the HDM.

Representing a Simple Relational Model

Relational Construct	**HDM Representation**
construct: `Rel` class: `nodal` scheme: $\langle\langle \mathsf{R} \rangle\rangle$	node: $\langle\langle \mathsf{R} \rangle\rangle$
construct: `Att` class: `link-nodal, constraint` scheme: $\langle\langle \mathsf{R}, \mathsf{a}, \mathsf{n} \rangle\rangle$	node: $\langle\langle \mathsf{R} : \mathsf{a} \rangle\rangle$ edge: $\langle\langle \_, \mathsf{R}, \mathsf{R} : \mathsf{a} \rangle\rangle$ if $\mathsf{n} = \mathsf{null}$ then constraint: $\langle \langle\langle \_, \mathsf{R}, \mathsf{R} : \mathsf{a} \rangle\rangle, \{0,1\}, \{1..N\} \rangle$ else constraint: $\langle \langle\langle \_, \mathsf{R}, \mathsf{R} : \mathsf{a} \rangle\rangle, \{1\}, \{1..N\} \rangle$

Table 3.1: Representing Simple Relational Model Constructs

We show in Table 3.1 how a simple relational data model can be represented in the HDM. In our simple relational model, there are two kinds of schema construct: Rel and Att. A Rel construct is identified by a scheme $\langle\langle \mathsf{R} \rangle\rangle$ where R is the relation name, and a Att construct is identified by a scheme $\langle\langle \mathsf{R}, \mathsf{a}, \mathsf{n} \rangle\rangle$ where a is an attribute (key or non-key) which may be null or $\mathsf{notnull}$ (denoted by n). In Table 3.1, we use some shorthand notation for expressing cardinality constraints on HDM edges, $\langle \langle\langle \mathsf{name}, \mathsf{c_1}, ..., \mathsf{c_m} \rangle\rangle, s_1, ..., s_m \rangle$, where name is the edge name which can be anonymous (denoted by "_"), each $\mathsf{c_i}$ is a participating construct in the edge, and each s_i is a set of integers representing the possible values for the cardinality of each $\mathsf{c_i}$ in the edge. Note that N denotes infinity in the table. The extent of a Rel construct $\langle\langle \mathsf{R} \rangle\rangle$ in an AutoMed relational schema is the projection of the relation $R(k_1, ..., k_n, a_1, ..., a_m)$ onto its primary key attributes $k_1, ..., k_n$. The extent of each Att construct $\langle\langle \mathsf{R}, \mathsf{a}, \mathsf{n} \rangle\rangle$ of R is the projection of R onto $k_1, ..., k_n, a$, where $a \in k_1, ..., k_n, a_1, ..., a_m$.

For example, a relation $\mathsf{student}(\underline{\mathsf{id}}, \mathsf{sex}, \mathsf{dname})$ would be modeled by a Rel construct $\langle\langle \mathsf{student} \rangle\rangle$ and three Att constructs $\langle\langle \mathsf{student}, \mathsf{id}, \mathsf{notnull} \rangle\rangle$, $\langle\langle \mathsf{student}, \mathsf{sex}, \mathsf{null} \rangle\rangle$ and $\langle\langle \mathsf{student}, \mathsf{dname}, \mathsf{notnull} \rangle\rangle$. Note that, for ease of exposition, in this thesis we may omit the n notation in Att constructs and we do not consider the null feature of attributes, so that the above three Att constructs are simplified into $\langle\langle \mathsf{student}, \mathsf{id} \rangle\rangle$, $\langle\langle \mathsf{student}, \mathsf{sex} \rangle\rangle$ and $\langle\langle \mathsf{student}, \mathsf{dname} \rangle\rangle$. We also ignore primary keys and foreign keys and refer the reader to [MP99b] for an encoding of a richer relational data model, including the modelling of constraints.

Representing a Simple XML Model

Table 3.2 shows the representation of a simple XML model in terms of the HDM. In this model, there are three kinds of schema construct: Element, Attribute and NestSet. The extent of an Element construct $\langle\langle \mathsf{e} \rangle\rangle$ consists of all the elements with tag e in the XML document; the extent of each Attribute construct $\langle\langle \mathsf{e}, \mathsf{a} \rangle\rangle$ consists of all pairs of elements and attributes x, y such that element x has tag e and has an attribute a with value y; and the extent of each NestSet construct $\langle\langle \mathsf{e_p}, \mathsf{e_c} \rangle\rangle$ consists of all pairs of elements x, y such that element x has tag p and has a child element y with tag c. We refer the reader to [Zam04] for an encoding of a richer model for XML data sources, called XML DataSource Schemas

(XMLDSS), which also captures the ordering of children elements under parent elements. That paper gives an algorithm for generating the XMLDSS of an XML document. That paper also discusses a unique naming scheme for Element constructs and their instances. In particular, $\langle elementName \rangle \$ \langle count \rangle$ is used for Element constructs, where $\langle count \rangle$ is a counter incremented every time the same $\langle elementName \rangle$ is encountered in a depth-first traversal of the schema; and $\langle elementName \rangle \$ \langle count \rangle \_ \langle instance \rangle$ is used for instances of an Element construct where $\langle instance \rangle$ is a counter incremented every time a new instance of the corresponding schema element is encountered in the document. If the $\$\langle count \rangle$ is omitted from an element name, then \$1 is assumed.

XML Construct	**Equivalent HDM Representation**
Construct: Element Class nodal Scheme $\langle\langle \mathsf{e} \rangle\rangle$	node: $\langle\langle \mathsf{xml : e} \rangle\rangle$
Construct: Attribute Class: link-nodal, and constraint Scheme: $\langle\langle \mathsf{e, a} \rangle\rangle$	node: $\langle\langle \mathsf{xml : e : a} \rangle\rangle$ edge: $\langle\langle \_, \mathsf{xml : e, xml : e : a} \rangle\rangle$ links: $\langle\langle \mathsf{xml : e} \rangle\rangle$ constraint: $\langle \langle\langle \_, \mathsf{xml : e, xml : e : a} \rangle\rangle, \{0, 1\}, \{1..N\} \rangle$
Construct NestSet Class link, constraint Scheme $\langle\langle \mathsf{e_p, e_c} \rangle\rangle$	edge: $\langle\langle \_, \mathsf{xml : e_p, xml : e_c} \rangle\rangle$ links: $\langle\langle \mathsf{xml : e_p} \rangle\rangle$, $\langle\langle \mathsf{xml : e_c} \rangle\rangle$ constraint: $\langle \langle\langle \_, \mathsf{xml : e_p, xml : e_c} \rangle\rangle, \{0..N\}, \{1\} \rangle$

Table 3.2: Representing Simple XML Model Constructs

To illustrate, Figure 3.2 shows a XML file which is modeled by three Element constructs, four Attribute constructs, and two NestSet constructs.

The Element constructs and their extents are as follows, where $[\ldots]$ denotes a list in IQL, $\{\ldots\}$ denotes a tuple (in this case one-tuples), and $'\ldots'$ denotes a string in IQL:

$$\langle\langle \mathsf{root} \rangle\rangle = [\texttt{'root\_1'}]$$

$$\langle\langle \mathsf{course} \rangle\rangle = [\texttt{'course\_1'}, \texttt{'course\_2'}]$$

$$\langle\langle \mathsf{student} \rangle\rangle = [\texttt{'student\_1'}, \texttt{'student\_2'}, \texttt{'student\_3'}, \texttt{'student\_4'}]$$

```
<?XML version='1.0'? >
<root>
  <course CID =“ISC01” cname =“Math”>
      <student SID =“ISS01” mark =“76”/ >
      <student SID =“ISS02” mark =“78”/ >
  < /course>
  <course CID =“ISC02” cname =“Programming”>
      <student SID =“ISS01” mark =“86”/ >
      <student SID =“ISS02” mark =“85”/ >
  < /course>
< /root>
```

Figure 3.2: A XML File

The Attribute constructs and their extents are as follows:

$$\begin{aligned}
\langle\langle \mathsf{course}, \mathsf{CID} \rangle\rangle &= [\{'\texttt{course\_1}', '\texttt{ISC01}'\}, \{'\texttt{course\_2}', '\texttt{ISC02}'\}] \\
\langle\langle \mathsf{course}, \mathsf{cname} \rangle\rangle &= [\{'\texttt{course\_1}', '\texttt{Math}'\}, \{'\texttt{course\_2}', '\texttt{Programming}'\}] \\
\langle\langle \mathsf{student}, \mathsf{SID} \rangle\rangle &= [\{'\texttt{student\_1}', '\texttt{ISS01}'\}, \{'\texttt{student\_2}', '\texttt{ISS02}'\}, \\
&\quad \{'\texttt{student\_3}', '\texttt{ISS01}'\}, \{'\texttt{student\_4}', '\texttt{ISS02}'\}] \\
\langle\langle \mathsf{student}, \mathsf{mark} \rangle\rangle &= [\{'\texttt{student\_1}', 76\}, \{'\texttt{student\_2}', 78\}, \\
&\quad \{'\texttt{student\_3}', 86\}, \{'\texttt{student\_4}', 85\}]
\end{aligned}$$

The two NestSet constructs and their extent are:

$$\begin{aligned}
\langle\langle \mathsf{course}, \mathsf{student} \rangle\rangle &= [\{'\texttt{course\_1}', '\texttt{student\_1}'\}, \{'\texttt{course\_1}', '\texttt{student\_2}'\}, \\
&\quad \{'\texttt{course\_2}', '\texttt{student\_3}'\}, \{'\texttt{course\_2}', '\texttt{student\_4}'\}] \\
\langle\langle \mathsf{root}, \mathsf{student} \rangle\rangle &= [\{'\texttt{root\_1}', '\texttt{course\_1}'\}, \{'\texttt{root\_1}', '\texttt{course\_2}'\}]
\end{aligned}$$

Representing a Simple Multidimensional Model

Our simple multidimensional data model has four kinds of schema construct: Fact, Dim (dimension), Att (non-key attribute) and Hierarchy. For simplicity, we model a measure as any other non-key attribute. Fact and Dim are nodal constructs, Att is a link-nodal construct and Hierarchy is a constraint. This specification is illustrated in Table 3.3.

A fact or dimension table R with primary attributes $\mathsf{k}_1, \ldots, \mathsf{k}_\mathsf{n}$ ($\mathtt{n} \geq 1$) is uniquely identified by the scheme $\langle\langle \mathsf{R}, \mathsf{k}_1, \ldots, \mathsf{k}_n \rangle\rangle$. This translates in the HDM into a nodal construct $\langle\langle \mathsf{R} \rangle\rangle$ the extent of which is the projection of the table R onto its primary key attributes $\mathsf{k}_1, \ldots, \mathsf{k}_n$. Each non-key attribute a of a fact or dimension table R is uniquely identified by the scheme $\langle\langle \mathsf{R}, \mathsf{a} \rangle\rangle$. This translates in the HDM into a link-nodal construct comprising a new node $\langle\langle \mathsf{R} : \mathsf{a} \rangle\rangle$ and an edge $\langle\langle \_, \mathsf{R}, \mathsf{R} : \mathsf{a} \rangle\rangle$. The extent of the edge is the projection of table R onto $\mathsf{k}_1, \ldots, \mathsf{k}_n, \mathsf{a}$.

Hierarchy constructs reflect the relationship between a primary key attribute k_i in a fact table R and its referenced foreign key attribute k'_j in a dimension table R', or between a primary key attribute in a dimension table R and its referenced foreign key attribute in a sub-dimension table R'. A hierarchy construct maps to a constraint in the corresponding HDM schema, which asserts that the set of values of k_i in R are always contained in the set of values for k'_j in R'.

Dimensional Construct	**HDM Representation**
construct: `Fact` class: `nodal` scheme: $\langle\langle \mathsf{R}, \mathsf{k}_1, \ldots, \mathsf{k}_n \rangle\rangle$	node: $\langle\langle \mathsf{R} \rangle\rangle$
construct: `Dim` class: `nodal` scheme: $\langle\langle \mathsf{R}, \mathsf{k}_1, \ldots, \mathsf{k}_n \rangle\rangle$	node: $\langle\langle \mathsf{R} \rangle\rangle$
construct: `Att` class: `link-nodal` scheme: $\langle\langle \mathsf{R}, \mathsf{a} \rangle\rangle$	node: $\langle\langle \mathsf{R} : \mathsf{a} \rangle\rangle$ edge: $\langle\langle \_, \mathsf{R}, \mathsf{R} : \mathsf{a} \rangle\rangle$
construct: `Hierarchy` class: `constraint` scheme: $\langle\langle \mathsf{R}, \mathsf{R}', \mathsf{k}_i, \mathsf{k}'_j \rangle\rangle$	constraint: $[\mathsf{x}_i \mid \{\mathsf{x}_1, \ldots, \mathsf{x}_n\} \leftarrow \langle\langle \mathsf{R} \rangle\rangle] \subseteq$ $[\mathsf{y}_j \mid \{\mathsf{y}_1, \ldots, \mathsf{y}_m\} \leftarrow \langle\langle \mathsf{R}' \rangle\rangle]$

Table 3.3: Representing Simple Multidimensional Model Constructs

3.2.2 The IQL Query Language

AutoMed supports a functional query language as its intermediate query language (IQL)[3]. IQL is a comprehensions-based functional query language. Such languages subsume query languages such as SQL and OQL in expressiveness [Bun94]. References [JPZ03, Pou04] give the details of IQL and references to other work on comprehension-based functional query languages. Here, we give an overview of IQL to the level of detail necessary for this thesis.

IQL supports several primitive operators for manipulating lists. The list append operator, `++`, concatenates two lists together. The `distinct` operator removes duplicates from a list and the `sort` operator sorts a list. The *monus* operator [Alb91], $--$, takes two lists and subtracts each member of the second list from the first e.g. [1,2,3,2,4]--[4,4,2,1] = [3,2]. The `fold` operator applies a given function `f` to each element of a list and then 'folds' a binary operator `op` into the resulting values. It is defined recursively as follows, where `(x:xs)` denotes a list with head `x` and tail `xs`:

```
fold f op e [] = e
fold f op e (x:xs) = (f x) op (fold f op e xs)
```

Other IQL list manipulation operators can be specified using `fold` together with IQL's support of `lambda` abstractions and set of built-in arithmetic and boolean operators (such as $+, -, *, /, >, <, =, !=, >=, <=$, and, or, not, member)[4]. For example, the IQL functions `sum` and `count` are equivalent to SQL's SUM and COUNT aggregation functions and can be specified as

[3]IQL is an "intermediate" language because, in a virtual integration scenario, queries using the high-level query language supported by a global schema are translated into IQL queries over the schema constructs defined in AutoMed, and these IQL queries are then translated into the queries using the high-level query languages supported by the data sources so that they can be evaluated in the data sources.

[4]Although they can be specified in this way, for efficiency purposes, they are actually built-into the IQL Query Evaluator.

```
sum xs = fold (id) (+) 0 xs
count xs = fold (lambda x.1) (+) 0 xs
```

We also have

```
min xs = fold (id) lesser maxNum xs
max xs = fold (id) greater minNum xs
avg xs = let {s,c} = fold (lambda x.{x,1}) combine {0,0} xs
         in (s/c)
```

assuming constants `maxNum` and `minNum` and the following functions `lesser`, `greater` and `combine`:

```
greater = lambda x.lambda y.if (x > y) then x else y
lesser  = lambda x.lambda y.if (x < y) then x else y
combine = lambda {s1,c1}.lambda {s2,c2}.{s1+s2,c1+c2}
```

The function `flatmap` applies a list-valued function `f` to each member of a list `xs` and is defined in terms of `fold`:

```
flatmap f xs = fold f (++) [] xs
```

`flatmap` can in turn be used to specify selection, projection and join operators. For example, the `map` function is a generalised projection operator and is defined as

```
map f xs = flatmap (lambda x.[f x]) xs
```

`flatmap` can also be used to define *comprehensions* [Bun94]. For example, the following comprehension iterates through a list of students and returns those students who are not members of staff:

```
[x | x <- <<student>>; not (member <<staff>> x)]
```

and it translates into:

```
flatmap (lambda x.if (not (member <<staff>> x))
                  then [x] else []) <<student>>
```

$[\mathtt{e}|\mathtt{Q}_1;\ldots;\mathtt{Q}_n]$ is the general syntax of a comprehension, in which `e` is any well-typed IQL expression, and $\mathtt{Q}_1$ to $\mathtt{Q}_n$ are *qualifiers*, each qualifier being either a *filter* or a *generator*. A generator has syntax $\mathtt{p} \leftarrow \mathtt{E}$, where `p` is a *pattern* and `E` is a collection-valued expression. A pattern is an expression involving tuples, variables and constants only. A filter is a boolean-valued expression.

Grouping operators are also definable in terms of `fold` (see [PS97]). In particular, the operator `group` takes as an argument a list of pairs `xs` and groups them on their first component, while `gc aggFun xs` groups a list of pairs `xs` on their first component and then applies the aggregation function `aggFun` to the second component.

Although IQL is list-based, if the ordering of elements within lists is ignored then its operators are faithful to the expected bag semantics, and in this thesis henceforth we do assume bag semantics. Use of the distinct operator can be used to obtain set semantics if needed.

3.2.3 Transformation Pathways

As described in Section 3.2.1, each modelling construct of a higher-level modelling language can be specified as some combination of HDM nodes, edges and constraints. For any modelling language $\mathcal{M}$ specified in this way, AutoMed automatically provides a set of primitive schema transformations that can be applied to schema constructs expressed in $\mathcal{M}$. In particular, for every extensional construct of $\mathcal{M}$ there is an add and a delete primitive transformation which add and delete the construct into and from a schema. Such a transformation is accompanied by an IQL query specifying the extent of the added or deleted construct in terms of the rest of the constructs in the schema. For those constructs of $\mathcal{M}$ which have textual names, there is also a rename primitive transformation. Also available are contract and extend transformations which behave in the same way as add and delete except that they indicate that their accompanying query may only partially construct the extent of the new/removed schema construct. The contract and extend transformations can also take a pair of queries (lq, uq) specifying a lower and upper bound on the extent of the new/removed construct, instead of just one lower-bound query as described above. However, for the purpose of data integration in a warehousing environment, we typically require just the single-query versions of these transformations.

In more detail, the full set of primitive transformations for an extensional construct `T` of a modelling language $\mathcal{M}$ is as follows[5]:

- $\texttt{addT}(\texttt{c}, \texttt{q})$ applied to a schema S produces a new schema S' that differs from S in having a new `T` construct identified by the scheme `c`. The extent of `c` is given by query `q` on schema S.

- $\texttt{extendT}(\texttt{c}, \texttt{ql}, \texttt{qu})$ applied to a schema S produces a new schema S' that differs from S in having a new `T` construct identified scheme `c`. The minimum extent of `c` is given by query `ql`, which may take the constant value Void if no lower bound for this extent may be derived from S. The maximum extent of `c` is given by query `qu`, which may take the constant value Any if no upper bound for this extent may be derived from S.

- $\texttt{delT}(\texttt{c}, \texttt{q})$ applied to a schema S produces a new schema S' that differs from S in not having a `T` construct identified by `c`. The extent of `c` may be recovered by evaluating query `q` on schema S'.

  Note that $\texttt{delT}(\texttt{c}, \texttt{q})$ applied to a schema S producing schema S' is equivalent to $\texttt{addT}(\texttt{c}, \texttt{q})$ applied to S' producing S.

[5]For non-extensional constructs (*i.e.* constructs that map into HDM constraints) there are add, delete and rename transformations if the construct is named. In this thesis we do not consider constraint constructs because our major issues addressed, incremental view maintenance and data lineage tracing, only relate to extensional constructs. We assume that any constraints between the source data and the global data are satisfied.

- `contractT(c, ql, qu)` applied to a schema S produces a new schema S' that differs from S in not having a `T` construct identified by `c`. The minimum extent of `c` is given by query `ql`, which may take the constant value Void if no lower bound for this extent may be derived from S'. The maximum extent of `c` is given by query `qu`, which may take the constant value Any if no upper bound for this extent may be derived from S'.

  Note that `contractT(c, ql, qu)` applied to a schema S producing schema S' is equivalent to `extendT(c, ql, qu)` applied to S' producing S.

- `renameT(c, c')` applied to a schema S produces a new schema S' that differs from S in not having a `T` construct identified by scheme `c` and instead a `T` construct identified by scheme `c'` differing from `c` only in its name.

  Note that `renameT(c, c')` applied to a schema S producing schema S' is equivalent to `renameT(c', c)` applied to S' producing S.

For example, the set of primitive transformations for schemas expressed in the simple relational data model we defined in Section 3.2.1 is `addRel`, `extendRel`, `delRel`, `contractRel`, `renameRel`, `addAtt`, `extendAtt`, `delAtt`, `contractAtt` and `renameAtt`; and for schemas expressed in the simple XML model is `add-Element`, `extendElement`, `delElement`, `contractElement`, `renameElement`, `add-Attribute`, `extendAttribute`, `delAttribute`, `contractAttribute`, `renameAtt-ribute`, `addNestSet`, `extendNestset`, `delNestset` and `contractNestset`.

The queries present within transformations mean that each primitive transformation t has an automatically derivable *reverse transformation*, t^{-1}. In particular, each add/extend transformation is reversed by a delete/contract transformation with the same arguments, while each rename transformation is reversed by swapping its two arguments. Thus, AutoMed is a **both-as-view** (BAV) data integration system. As discussed in [MP03a], BAV subsumes the global-as-view

(GAV) and local-as-view (LAV) approaches [Len02], since it is possible to extract a definition of each global schema construct as a view over source schema constructs, and it is also possible to extract definitions of source schema constructs as views over the global schema. We refer the reader to [JTMP04] for details of AutoMed's GAV and LAV view generation algorithms.

In AutoMed, schemas are incrementally transformed by applying to them a sequence of primitive transformations $t_1, \ldots, t_r$. Each primitive transformation adds, deletes or renames just one schema construct. Thus, intermediate schemas may contain constructs of more than one modelling language.

We term a sequence of primitive transformations from one schema S_1 to another schema S_2 a transformation *pathway* from S_1 to S_2, denoted $S_1 \rightarrow S_2$. All source, intermediate and integrated schemas and the pathways between them are stored in AutoMed's Schemas & Transformations Repository (see Section 3.2.4 below).

The queries within transformations are used by AutoMed's Global Query Processor (GQP) [JPZ03] to evaluate an IQL query over a global schema in the case of a virtual data integration scenario. The GAV view definition for each global schema construct (*i.e.* the view definition over the source schema constructs) is derivable from the transformation pathways by using the view generation algorithm described in [JTMP04]. This algorithm traverses the transformation pathways from the source schemas to the global schema backwards, and unfolds any virtual schema construct using the query in the transformation step which created the construct, until all constructs in the unfolded query are materialised.

The process of evaluating a query over a virtual global schema includes: *Query Reformulation*, replacing the virtual global schema constructs in the query by their GAV view definitions; *Query Optimisation*, optimising the query by eliminating redundant parts of the query, and reorganising the query by gathering together the query parts which can be translated by the same data source so that bigger sub-queries can be sent to each data source wrapper to evaluate; *Query Annotation*, annotating the query by indicating which sub-queries have to be sent to which data sources; and *Query Evaluation*, communicating with data source wrappers by sending them sub-queries to evaluate, receiving the results, and undertaking any further necessary evaluation to obtain the final query result[6].

In the case that the global schema is materialised, the Query Evaluator can be used directly on the materialised data.

3.2.4 The AutoMed Metadata Repository

The AutoMed Metadata Repository forms a platform for other components of the AutoMed Software Architecture (illustrated in Figure 3.3) to be implemented upon. When a data source is wrapped, a definition of the schema for that data source is added to the repository. AutoMed's wrappers are implemented at two levels. A *high level wrapper* converts between AutoMed queries and data and the standard representation for a class of data sources *e.g.* the SQL92Wrapper converts between IQL and SQL92. A *low level wrapper* deals with differences between the class standard and a particular data source *e.g.* the PostgresSQLWrapper converts between SQL92 and Postgres databases.

The schema matching tool may be used to identify related objects in various data sources (accessing the query processor to retrieve data from schema objects) [Riz04]. After a schema matching phase, the schema restructuring tool can be applied to generate a transformation pathway from a source schema to the global

[6]As well as working on the data warehousing aspects of AutoMed, I have also contributed to the design and development of the query optimiser.

schema [ZP04]. The global query processor undertakes the processing described in the previous section, and includes the query reformulation, optimisation, annotation and evaluation processes. A GUI is supplied with AutoMed for these components, and it is possible for a user application to be configured to run from this GUI, and use the APIs of the various components. We focus here on using the AutoMed Metadata Repository in a data warehousing environment.

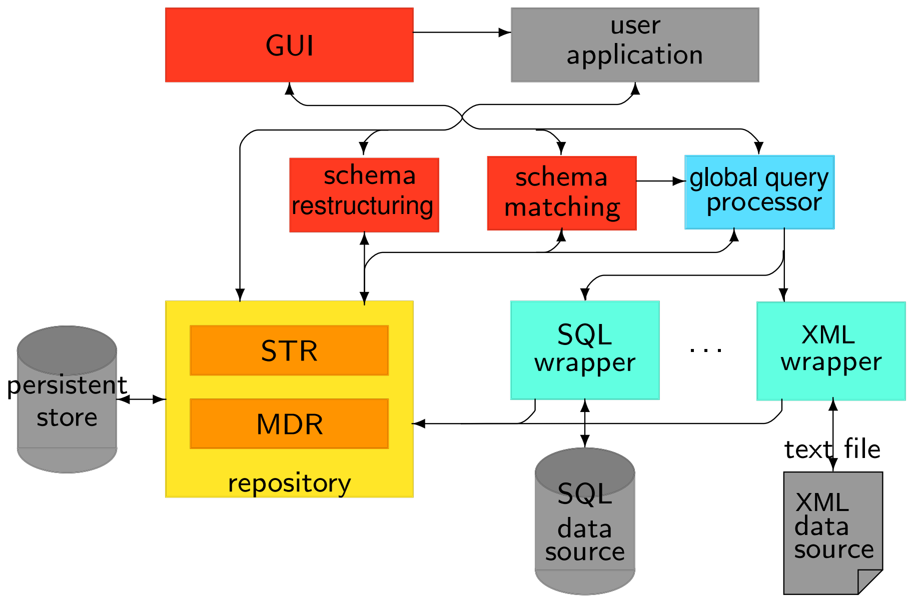


Figure 3.3: AutoMed Software Architecture

The repository has two logical components. The Model Definitions Repository (MDR) defines how each construct of a data modelling language is represented as a combination of nodes, edges and constraints in the HDM. The MDR is used to configure AutoMed so that it can handle a particular data modelling language. The Schemas and Transformations Repository (STR) defines schemas in terms of the data modelling constructs in the MDR, and transformations to be specified between such schemas. The MDR and STR may be held in the same or separate persistent storage. If the MDR and STR are stored in separate storage, many

AutoMed users can share a single MDR repository, which once configured, need not be updated when integrating data sources that conform to a known set of data modelling languages.

The API to these repositories uses JDBC to access an underlying relational database. Thus, these repositories can be implemented using any DBMS supporting JDBC. If the DBMS of the data warehouse supports JDBC, then the AutoMed repositories can be part of the data warehouse itself.

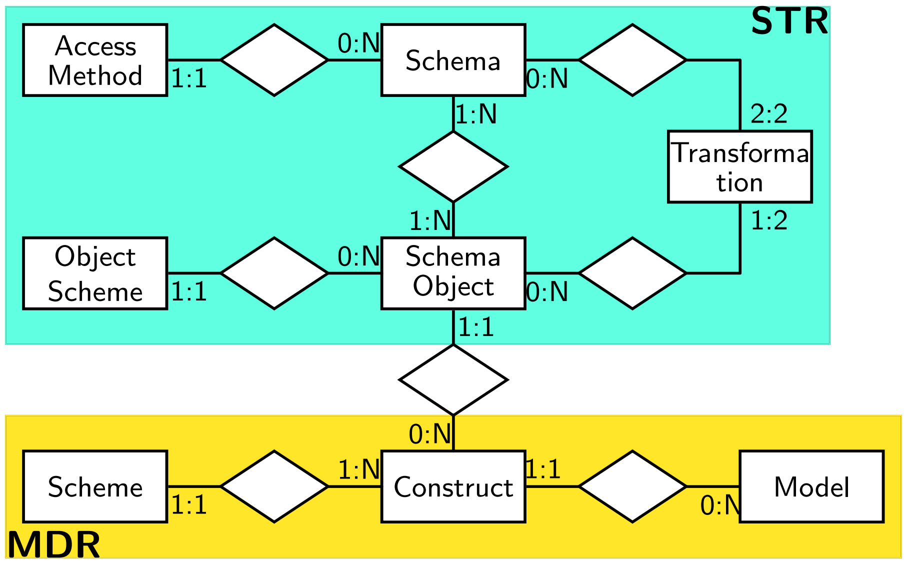


Figure 3.4: AutoMed Repository Schema

Figure 3.4 (taken from [BMT02]) gives an overview of the key objects in the repository. The STR contains a set of descriptions of Schemas, each of which contains a set of SchemaObject instances, each of which must be based on a Construct instance that exists in the MDR. This Construct describes how the SchemaObject can be constructed in terms of strings and references to other schema objects, and the relationship of the construct to the HDM. Schemas may be related to each other using instances of Transformation.

The AutoMed repository API provides methods to create, query, alter and remove models, constructs, schemas, schema objects and transformations. The repository API comprises of Java classes representing each of these entities and the methods for manipulating them[7].

3.3 Expressing Data Warehouse Schemas and Transformations

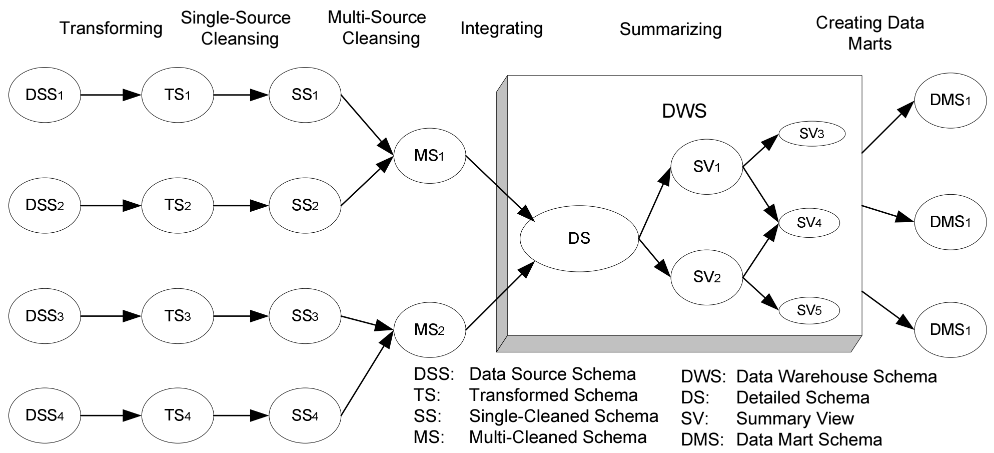


Figure 3.5: Data Transformation and Integration at the Schema Level

Figure 3.5 illustrates at the schema level the data transformation and integration processes in a typical data warehouse. Generally, the extract-transform-load (ETL) process of a data warehouse includes extracting data from the remote data sources into the staging area, cleansing and transforming data in the staging area and loading them into the data warehouse. In this section we assume that data extraction has already happened *i.e.* all the data sources are in the staging area. The data source schemas ($\texttt{DSS}_i$ in Figure 3.5) may be expressed in any modelling language that has been specified in AutoMed. The transforming process translates each $\texttt{DSS}_i$ into a transformed schema $\texttt{TS}_i$ which is ready for single-source

[7]For details, see `http://www.doc.ic.ac.uk/automed/resources/apidocs/index.html`

data cleansing. Each $\mathtt{TS}_i$ may be defined in the same, or a different, modelling language as $\mathtt{DSS}_i$ and other $\mathtt{TS}$s. The translation from a $\mathtt{DSS}_i$ to a $\mathtt{TS}_i$ is expressed as an AutoMed transformation pathway $\mathtt{DSS}_i \rightarrow \mathtt{TS}_i$. Such translation may not be necessary if the data cleansing tools to be employed can be applied directly to $\mathtt{DSS}_i$, in which case $\mathtt{TS}_i$ and $\mathtt{DSS}_i$ are identical.

The single-source data cleansing process transforms each $\mathtt{TS}_i$ into a single-source-cleansed schema $\mathtt{SS}_i$, which is defined in the same modelling language as $\mathtt{TS}_i$ but may be a different from it. The single-source cleansing process is expressed as an AutoMed transformation pathway $\mathtt{TS}_i \rightarrow \mathtt{SS}_i$. Multi-source data cleansing removes conflicts between sets of single-source-cleansed schemas and creates a multi-source-cleansed schema $\mathtt{MS}_i$ from them. Between the single-source-cleansed schemas and the detailed schema ($\mathtt{DS}$) of the data warehouse there may be several stages of $\mathtt{MS}$s, possibly represented in different modelling languages.

In general, if during multi-source data cleansing n schemas $S_1, \ldots, S_n$ need to be transformed and integrated into one schema S, we can first automatically create a 'union' schema $S_1 \cup \ldots \cup S_n$ (after first undertaking any renaming of constructs necessary to avoid any naming ambiguities between constructs from different schemas). We can then express the transformation and integration process as a pathway $S_1 \cup \ldots \cup S_n \rightarrow S$[8]. (There are also other schema integration approaches possible with AutoMed. With this approach, and in a data warehousing context, there is no need for `extend` transformation steps).

After multi-source data cleansing, the resulting $\mathtt{MS}$s are then transformed and integrated into a single detailed schema, $\mathtt{DS}$, expressed in the data model of the data warehouse. First, a union schema $\mathtt{MS}_1 \cup \ldots \cup \mathtt{MS}_n$ is automatically generated.

[8]Reference [AMGF05] is concerned with correlating data from different databases and provides semantically rich materialisation rules handling schema heterogeneity among the databases. The integrated schema can use one of the integration rules, such as union, merge and intersection, to integrate the source databases. This functionality can also be obtained using AutoMed, within the pathway $S_1 \cup \ldots \cup S_n \rightarrow S$.

The transformation and integration process is then expressed as a pathway $\mathtt{MS}_1 \cup \ldots \cup \mathtt{MS}_n \rightarrow \mathtt{DS}$. The `DS` can then be enriched with summary views by means of a transformation pathway from `DS` to the final data warehouse schema `DWS`.

Data mart schemas (`DMS`) can subsequently be derived from the `DWS` and these may be expressed in the same, or a different, modelling language as the `DWS`. Again, the derivation is expressed as a transformation pathway $\mathtt{DWS} \rightarrow \mathtt{DMS}$.

Using AutoMed, four steps are needed in order to create the metadata expressing the above schemas and transformation pathways:

1. **Create AutoMed repositories:** AutoMed metadata is stored in the MDR and the STR. So we first need to create these repositories including empty relations defined by the MDR and STR schemas illustrated in Figure 3.4.

2. **Specify data models:** All the data models that will be required for expressing the various schemas of Figure 3.5 need to be specified in terms of AutoMed's HDM, via the API of the MDR (standard definitions of relational, ER and XML data models are available).

3. **Extract data source schemas:** Each data source schema is automatically extracted and translated into its equivalent AutoMed representation using the appropriate wrapper for that data source.

4. **Define transformation pathways:** The remaining schemas of Figure 3.5 and the pathways between them can now be defined, via the API of the STR.

   After any primitive transformation is applied to a schema, a new schema results. By default, this will be an *intentional* schema within the STR *i.e.* it is not stored but its definition can be derived by traversing the pathway from its nearest ancestor *extensional* schema. The data source schemas are, by definition, extensional schemas *i.e.* their full definition is stored within the STR. It is also possible to request that any other schema becomes an extensional one, for example the successive stages of schemas identified in Figure 3.5.

After any addT(c,q) transformation step, it is possible to *materialise* the new construct c by creating, externally to AutoMed, a new data source whose schema includes c and populating this data source by the result of evaluating the query q (we discuss this process in more detail in Section 3.4.1 below). In general, a schema may be a *materialised* schema (all of its constructs are materialised) or a *virtual* schema (none of its constructs are materialised) or *partially materialised* (some of its constructs are materialised, some not).

In the following sections, we discuss in more detail how AutoMed transformation pathways can be used for describing the six stages of the data transformation and integration process illustrated in Figure 3.5. We first give a simple example illustrating data transformation and integration, assuming that no data cleansing is necessary.

3.3.1 An Example of Data Integration and Transformation

Figure 3.6 shows a multidimensional schema consisting of a fact table Salary and two dimension tables Person and Job, which is represented by AutoMed schema constructs ⟨⟨Salary, id, job_id⟩⟩, ⟨⟨Salary, salary⟩⟩, ⟨⟨Salary, dept_id⟩⟩, ⟨⟨Person, id⟩⟩, ⟨⟨Person, name⟩⟩, ⟨⟨Job, job_id⟩⟩, ⟨⟨Job, job_descr⟩⟩, ⟨⟨Salary, Person, id, id⟩⟩ and ⟨⟨Salary, Job, job_id, job_id⟩⟩; a XML schema consisting of elements root and dept and two attributes id and name, which is represented by AutoMed schema

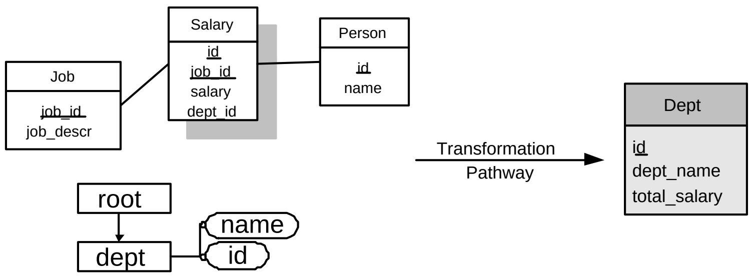


Figure 3.6: An Example of Data Integration and Transformation

constructs ⟨⟨root⟩⟩, ⟨⟨dept⟩⟩, ⟨⟨dept, id⟩⟩, ⟨⟨dept, name⟩⟩ and ⟨⟨root, dept⟩⟩; and a relational schema consisting of a single table Dept into which the other two schemas need to be transformed and integrated, which is represented by AutoMed schema constructs ⟨⟨Dept⟩⟩, ⟨⟨Dept, id⟩⟩, ⟨⟨Dept, dept_name⟩⟩ and ⟨⟨Dept, total_salary⟩⟩.

In order to integrate the two source schemas into the target schema we first form their union schema. The following four primitive transformations are then applied to this union schema in order to add the Dept relation to it, defining the extent of its id key attribute to be obtained by the XML id attribute, the extent of its dept_name attribute to be obtained by the XML name attribute, and the extent of its total_salary attribute to be obtained by summing the salaries for each department in the Salary table:

```
addRel (⟨⟨Dept⟩⟩, map (lambda {k,i}.i) ⟨⟨dept, id⟩⟩);
addAtt (⟨⟨Dept, id⟩⟩, map (lambda {k,i}.{i,i}) ⟨⟨dept, id⟩⟩);
addAtt (⟨⟨Dept, dept_name⟩⟩, [{i,n}|{k,i} ← ⟨⟨dept, id⟩⟩; {k',n}← ⟨⟨dept, name⟩⟩;
                                    k=k']);
addAtt (⟨⟨Dept, total_salary⟩⟩, gc sum [{d,s}|{i,j,s}← ⟨⟨Salary, salary⟩⟩;
                                         {i',j',d}← ⟨⟨Salary, dept_id⟩⟩;
                                         i=i'; j=j']);
```

The following five transformations can then be applied to the resulting schema to remove the XML constructs from it — note how the queries show how the extents of these constructs could be reconstructed from the remaining schema constructs. In particular, IQL functions `generateUID s xs` and `generateAtt s xs` are used to generate instances of XML elements and attributes, where the input `s` is a string and `xs` is a list of n-tuples. The function `generateUID` generates a list of values of the form `s_count` in which `count` is a counter incremented every time a new value is generated. The number of values generated is equal to the number of items in the list `xs`. The function `generateAtt` generates a list of tuples of the form `{s_count,c1,...,cn}` in which `s` and `count` are as above and `{c0,c1,...,cn}` is a tuple in the list `xs`. For example, suppose `s` is `'dept'` and `xs` is [`{'D01','Sales'}`, `{'D02','Accounts'}`, `{'D03','Personnel'}`], the result of `generateUID s xs` is the list [`'dept_1'`, `'dept_2'`, `'dept_3'`], and the result of `generateAtt s xs` is the list [`{'dept_1','Sales'}`, `{'dept_2','Accounts'}`, `{'dept_3','personnel'}`].

```
delNestSet   (⟪root, dept⟫, [{'root_1',c} | c ← generateUID 'dept' ⟪Dept⟫]);
delAttribute (⟪dept, name⟫, generateAtt 'dept' ⟪Dept, dept_name⟫);
delAttribute (⟪dept, id⟫, generateAtt 'dept' ⟪Dept, dept_id⟫);
delElement   (⟪dept⟫, generateUID 'dept' ⟪Dept⟫);
delElement   (⟪root⟫, ['root_1']);
```

Finally, the following sequence of transformations remove the multidimensional schema constructs — note that contract rather than delete transformations are used since their extents cannot be reconstructed from the remaining schema constructs:

```
contractHierarchy (⟪Salary, Person, id, id⟫);
contractHierarchy (⟪Salary, Job, job_id, job_id⟫);
contractAtt       (⟪Salary, salary⟫);
contractAtt       (⟪Salary, dept_id⟫);
contractFact      (⟪Salary, id, job_id⟫);
```

```
contractAtt      (⟨⟨Job, job_descr⟩⟩);
contractDim      (⟨⟨Job, job_id⟩⟩);
contractAtt      (⟨⟨Person, name⟩⟩);
contractDim      (⟨⟨Person, id⟩⟩);
```

The final schema consists of the Dept relation and its attributes, as required.

This example illustrates how schemas expressed in one data model can be transformed into a schema expressed in another. The general approach is to first add the new schema constructs of the target data model (relational in the above example) and then to delete or contract the schema constructs of the original data model(s) (multidimensional and XML in the above example).

3.3.2 Expressing Data Cleansing

We recall from Chapter 2 that the problem of data cleansing includes single-source problems and multi-source problems, and that both of them have two levels, schema-level and instance-level. In this section, we investigate how AutoMed metadata can be used for expressing data cleansing processes, for both single and multiple data sources, and for both schema-level and instance-level problems.

Single-Source Cleansing

Schema-level single-source problems may arise within a transformed schema $\texttt{TS}_i$ in Figure 3.5 and they can be resolved by means of an AutoMed transformation pathway that evolves $\texttt{TS}_i$ as necessary.

Single-source instance-level problems include value, attribute and record problems. Value problems occur within a single value and include problems such as missing values, misspelled values, mis-fielded values, embedded values, mis-expressed values, or values using abbreviations. Attribute problems relate to multiple attributes in one record and include problems such as dependence violation. Record problems relate to multiple records in the data sources and include problems such as duplicate records or contradictory records.

Handling some instance-level problems does not require the schemas to be evolved, only the extent of one or more schema constructs to be corrected. In general, suppose that the extent of a schema construct `c` needs to be replaced by a new, cleansed, extent. We can do this using an AutoMed pathway as follows:

1. Add a new temporary construct `temp` to the schema, whose extent consists of the 'clean' data that is needed to generate the new extent of `c`. This clean data is derived from the extents of the existing schema constructs. This derivation may be expressed as an IQL query, or as a call to an 'external' function or, more generally, as an IQL query with embedded calls to external functions.

   (The IQL interpreter is easily extensible with new built-in functions, implemented in Java, and these may themselves call out to other external functions. If the extent of a new schema construct depends on calls to one or more external functions, then the new construct must be materialised. Otherwise, if the extent of a new construct is defined purely in terms of IQL and its own built-in functions then the new construct need not be materialised.)

2. Contract the construct `c` from the schema.

3. Add a new construct `c` whose extent is derived from `temp`.

4. Delete or contract the `temp` construct.

To illustrate, suppose we have available a built-in function toolCall which allows a specified external data cleansing tool to be invoked with specified input data. Then, we can invoke a data cleansing tool, for example "QuickAddress Batch"[9] to correct the zip and address attributes of a table Person(id, name, address, zip, city, country, phoneAndFax, maritalStatus) by regenerating these attributes given the combination of address, zip and city information:

```
addRel        (⟪Temp, id, address, zip⟫,
                   toolCall 'QuickAddress Batch' '⟪Person, address⟫'
                            '⟪Person, zip⟫ ' '⟪Person, city⟫');
contractAtt  (⟪Person, zip⟫);
contractAtt  (⟪Person, address⟫);
addAtt       (⟪Person, zip⟫, [{i,z}|{i,a,z} ← ⟪Temp, id, address, zip⟫]);
addAtt       (⟪Person, address⟫, [{i,a}|{i,a,z} ← ⟪Temp, id, address, zip⟫]);
delRel    (⟪Temp, id, address, zip⟫, [{i,a,z}|{i,a} ← ⟪Person, address⟫;
                                          {i',z}← ⟪Person, zip⟫; i = i']);
```

Handling some instance-level problems may require the schemas to be evolved. For example, if we have available a built-in function split_phone_fax which slits a string comprising a phone number followed by one or more spaces followed by a fax number into a pair of numbers, then the following AutoMed pathway converts the attribute phoneAndFax of the Person table above into two new attributes phone and fax:

```
addRel  (⟪Temp, id, phone, fax⟫, [{i,p,f}|{i,pf} ← ⟪Person, phoneAndFax⟫;
                                        {p,f} ← split_phone_fax pf]);
addAtt  (⟪Person, phone⟫, [{i,p}|{i,p,f}← ⟪Temp, id, phone, fax⟫]);
addAtt  (⟪Person, fax⟫, [{i,f}|{i,p,f}← ⟪Temp, id, phone, fax⟫]);
contractAtt  (⟪Person, phoneAndFax⟫);
delRel  (⟪Temp, id, phone, fax⟫, [{i,p,f}|{i,p}← ⟪Person, phone⟫;
                                        {i',f}← ⟪Person, fax⟫; i = i']);
```

[9]http://www.qas.com/address-correction-software.asp

Multi-Source Cleansing

After single-source data cleansing, there may still exist conflicts between different single-source cleansed schemas in Figure 3.5, leading to the process of multi-source cleansing.

Schema-level problems in multi-source cleansing include attribute and structure conflicts. Attribute conflicts arise when different sources use the same name for different constructs (homonyms) or different names for the same construct (synonyms), and they can be resolved by applying appropriate rename transformations to one of the schemas. Structure conflicts arise when the same information is modeled in different ways in different schemas, and they can be resolved by evolving one or more of the schemas using appropriate AutoMed pathways. For example, the transformation pathway in Section 3.3.1 shows how the department information modeled in an XML schema can be transformed into the equivalent information modeled in a simple relational schema.

Instance-level problems in multi-source cleansing include attribute, record, reference and data source problems. Attribute problems include different representations of the same attribute in different schemas or different interpretations of the values of an attribute in different schemas. Such problems can be resolved by generating a new extent for the attribute in one of the schemas by applying an appropriate conversion function to each of its values. In general, suppose we wish to convert each of the values within the extent of a construct `c` in a schema S by applying a function `f` to it. First a new construct `c_new` is added to S, whose extent is populated by iterating over the extent of `c` and applying `f` to each of its values. Then, the old construct `c` is deleted or contracted from the schema, and finally `c_new` is renamed to `c`. For example, the following pathway converts a 'M'/'S' representation for the maritalStatus attribute in the above Person table into a 'Y'/'N' representation, assuming the availability of a built-in function convertMS which maps 'M' to 'Y' and 'S' to 'N':

```
addAtt       (⟨⟨Person, maritalStatus_new⟩⟩,
                [{i,convertMS s}|{i,s} ← ⟨⟨Person, maritalStatus⟩⟩]);
contractAtt  (⟨⟨Person, maritalStatus⟩⟩);
renameAtt    (⟨⟨Person, maritalStatus_new⟩⟩, ⟨⟨Person, maritalStatus⟩⟩);
```

Note that if there is also available an inverse function convertMSinv which maps 'Y' to 'M' and 'N' to 'S', then a delete transformation could have been used in the second step above instead of a contract:

```
delAtt   (⟨⟨Person, maritalStatus⟩⟩,
              [{i,convertMSinv s}| {i,s}← ⟨⟨Person, maritalStatus_new⟩⟩]);
```

Record problems in multi-source cleansing include duplicate records or contradictory records among different data sources. For duplicate records, suppose that constructs c and c' from different schemas are to be integrated into a single construct within some multi-source cleansed schema. Then, prior to the integration, we can create a new extent for c comprising only those values not present in the extent of c':

```
add      (c_new, [v | v ← c; not (member c' v)];
contract (c);
rename   (c_new, c);
```

For contradictory records, we can similarly create a new extent for c comprising only those values which do not contradict values in the extent of c'. For example, suppose we have tables Person and Employee in different schemas, both with key id, and the attributes ⟨⟨Person, maritalStatus⟩⟩ and ⟨⟨Emp, maritalStatus⟩⟩ are going to be integrated into a single attribute of a single table within some multi-source cleansed schema. Then the following transformation removes values from ⟨⟨Person, maritalStatus⟩⟩ which contradict values in ⟨⟨Emp, maritalStatus⟩⟩ (assuming that the latter is the more reliable source — the opposite choice would also of course be possible[10]);

```
addAtt       (⟪Person, maritalStatus_new⟫,
              ⟪Person, maritalStatus⟫−−[{i,s}|{i,s} ← ⟪Person, maritalStatus⟫;
                                         {i',s'}← ⟪Emp, maritalStatus⟫;
                                         i = i'; not (s = s')]);
contractAtt  (⟪Person, maritalStatus⟫);
renameAtt    (⟪Person, maritalStatus_new⟫, ⟪Person, maritalStatus⟫);
```

Reference problems in multi-source cleansing occur when a referenced value does not exist in the target schema construct and can be resolved by removing the dangling references. For example, if an attribute ⟪Emp, dept_id⟫ references a table ⟪Dept⟫ with key ⟪dept_id⟫, then the following transformation removes values from ⟪Emp, dept_id⟫ for which there is no corresponding ⟪dept_id⟫ value in ⟪Dept⟫:

```
addAtt (⟪Emp, dept_id_new⟫, [{i,d}|{i,d}← ⟪Emp, dept_id⟫; member ⟪Dept⟫ d]);
contractAtt (⟪Emp, dept_id⟫);
renameAtt   (⟪Emp, dept_id_new⟫, ⟪Person, dept_id⟫);
```

Finally, data source problems relate to whole data sources, for example, aggregation at different levels of detail in different data sources (*e.g.* sales may be recorded per product in one data source and per product category in another data source). Such conflicts can be resolved either by retaining both sets of source data within the target multi-source schema $\texttt{MS}_i$ (with appropriate renaming of schema constructs as necessary) or by selecting the 'coarser' aggregation and creating a view over the more detailed data which summarises this data at the coarser level, ready for integration with the more coarsely aggregated data from the other data source.

[10]We could also use the taxonomy of quality defined in [BGF02] to decide which is the more reliable source.

3.3.3 Expressing Data Integration

After data cleansing, the resulting multi-source-cleansed schemas $\mathtt{MS}_1, \ldots, \mathtt{MS}_n$ are ready to be transformed and integrated into the detailed schema, DS, via the automatically generated union schema $\mathtt{MS}_1 \cup \ldots \cup \mathtt{MS}_n$. Section 3.3.1 above illustrated this process.

3.3.4 Expressing Data Summarisation

Data summarisation defines views over the detailed data. These are expressed by means of a transformation pathway from DS to the final data warehouse schema DWS, consisting of a series of add steps defining the new summarised constructs as views over the constructs of DS. The example in Section 3.3.1 illustrates a process of defining a summarised view over heterogeneous data sources.

3.3.5 Creating Data Marts

Data mart schemas (DMS) can subsequently be derived from the DWS, again by means of a transformation pathway $\mathtt{DWS} \rightarrow \mathtt{DMS}$. Unlike the previous, summarising, step the target schema may be expressed in a different modelling language to the DWS. In fact, this step can be regarded as a separate instance of Figure 3.5 where the DWS now plays the role of the (single) data source and the DMS plays the role of the target warehouse schema. The scenario is a simplification of Figure 3.5 since there is only one data source, and there are no single-source or multi-source cleansed schemas.

3.4 Using the Transformation Pathways

In the previous section we showed how AutoMed metadata can be used for expressing the processes of data transformation, cleansing, integration, summarisation and creating data marts in a data warehouse. In this section, we discuss how the resulting transformation pathways can be used for some key data warehousing processes: populating the data warehouse, incrementally maintaining the warehouse data after data source updates, and tracing the lineage of warehouse data.

3.4.1 Populating the Data Warehouse

In order to use the AutoMed transformation pathways for populating the data warehouse, an AutoMed wrapper is required for each kind of data store from which data will be extracted or into which data will be stored. In order to populate a construct `c` of the data warehouse schema `DWS`, we need to generate a view definition for each construct of `DWS` in terms of its nearest ancestor materialised constructs within the pathways from the data source schemas $\mathtt{DSS}_1, \ldots, \mathtt{DSS}_n$ to `DWS`. This can be done using a modification of the GAV view generation algorithm described in [JTMP04]. This algorithm traverses the pathway from `DWS` to each $\mathtt{DSS}_i$ backwards, all the way to $\mathtt{DSS}_i$. The modified algorithm stops whenever a materialised construct is encountered in a pathway. The result is a view definition of the construct `c` in terms of already materialised constructs. This view definition is an IQL query which can be evaluated, and the resulting data can be inserted into the data store linked with `c`, via a series of update requests to that data store's wrapper.

3.4.2 Incrementally Maintaining the Warehouse Data

In order to incrementally maintain materialised warehouse data, we need to use incremental view maintenance techniques. If a materialised construct c in the data warehouse schema DWS is defined by an IQL query q over other materialised constructs, we give in Chapter 7 formulae for incrementally maintaining c if one its ancestor materialised constructs c^{anc} has new data inserted into it (an increment) or data deleted from it (a decrement). We actually do not use the whole view definition q generated for c, but instead track the changes from c^{anc} through each step of the pathway to DWS. At each add or rename step we use the set of increments and decrements computed so far to compute the increment and decrement for the schema constructed being generated by this step of the pathway. Chapter 7 discusses this in detail.

3.4.3 Tracing the Lineage of the Warehouse Data

The *lineage* of a data item t in the extent of a materialised construct c of the warehouse schema DWS is a set of source data items from which t was derived. In Chapter 5, we develop definitions for data lineage in the context of AutoMed transformation pathways and give formulae for deriving the lineage of a data item t in the extent of a materialised construct c created by a transformation step of the form addT(c,q). We then give an algorithm for tracing the lineage of t all the way back to the data sources by using the AutoMed pathways from the data source schemas $\texttt{DSS}_1, \ldots, \texttt{DSS}_n$ to the warehouse schema. This algorithm traverses a pathway backwards, and incrementally computes new lineage data whenever an add or rename step is encountered, finally ending with the required lineage for t from within $\texttt{DSS}_1, \ldots, \texttt{DSS}_n$.

Chapter 6 generalises these algorithms to use arbitrary AutoMed schema transformations for tracing data lineage *i.e.* where intermediate schema constructs may or may not be materialised.

3.5 Discussion

In this chapter we have shown how AutoMed metadata can be used to express the processes of data transformation, cleansing, integration, summarisation and creating data marts in a heterogeneous data warehouse. In particular, for all categories of data cleansing problems, the general approach is to add new constructs to the current schema and to populate them by 'clean' data generated from the extents of the existing schema constructs by means of IQL queries and/or or calls to external functions. The old, 'dirty', schema constructs are then contracted from the schema. Compared with the commercial tools and general research tools for data cleansing discussed in Section 2.4.3 of Chapter 2, we express the process of data cleansing using a sequence of transformations which readily supports schema evolution (see points 2 and 3 below). Other data cleansing tools can be called from our data cleansing process via built-in functions within IQL queries. Furthermore, we consider data cleansing both at the schema level and at the instance level, while only one of these aspects is typically considered in other data cleansing tools.

We have also discussed how the resulting transformation pathways can be used for populating the data warehouse, incrementally maintaining the data warehouse data after data source updates, and tracing the lineage of data warehouse data. More detail about the latter two will be given in Chapters $5-7$ of the thesis. In this thesis we assume that the data warehouse is not updated directly but only based on periodic changes to the data sources in the staging area.

There are three main differences between our approach and the traditional data warehousing approach based on a single conceptual data model (CDM):

1. In the CDM approach, each data source wrapper translates the data source model into the CDM. Since both are likely to be high-level conceptual models, semantic mismatches may exist between the CDM and the source data model, and there may be a loss of information between them. In contrast, with our approach, the data source wrappers translate each data source schema into its equivalent AutoMed representation. Any necessary inter-model translation then happens explicitly within the AutoMed transformation pathways, under the control of the data warehouse designer.

2. In the CDM approach, the data transformation and integration metadata is tightly coupled with the CDM of the particular data warehouse. If the data warehouse is to be redeployed on a platform with a different CDM, it is not easy to reuse the previous data transformation and implementation effort. In contrast, with our approach it is possible to extend the existing pathways from the data source schemas $\mathtt{DSS}_1, \ldots, \mathtt{DSS}_n$ to the current detailed data warehouse schema, $\mathtt{DS}$, with extra transformation steps that evolve $\mathtt{DS}$ into a new schema $\mathtt{DS}^{new}$, expressed in the data model of the new data warehouse implementation. Chapter 4 discusses how in greater detail.

3. In the CDM approach, if a data source schema changes it is not straightforward to evolve the view definitions of the data warehouse constructs. With our approach, a change of a data source schema $\mathtt{DSS}_i$ into a new schema $\mathtt{DSS}_i^{new}$ can be expressed as a transformation pathway $\mathtt{DSS}_i \rightarrow \mathtt{DSS}_i^{new}$. The (automatically derivable) reverse pathway $\mathtt{DSS}_i^{new} \rightarrow \mathtt{DSS}_i$ can then be prefixed to the original pathway $\mathtt{DSS}_i \rightarrow \mathtt{TS}_i$ to give a pathway $\mathtt{DSS}_i^{new} \rightarrow \mathtt{TS}_i$, thus extending the transformation network of Figure 3.5 to encompass the

new schema. Chapter 4 discusses in greater detail the modifications to the transformation network and the change propagation process.

Chapter 4

Using AutoMed Transformation Pathways for Handling Schema Evolution

4.1 Motivation

The heterogeneity of the data sources of data warehouses has two aspects, heterogeneous data expressed in different data models, called *model heterogeneity* [KR02], and heterogeneous data within different data schemas expressed in the same data model, called *schema heterogeneity* [KR02, Mil98].

As we discussed in Chapter 3, the common approach to handling model heterogeneity is to use a single conceptual data model (CDM) for the data transformation and integration. Each data source has a wrapper for translating its schema and data into the CDM. The warehouse schema is derived from these CDM schemas by means of view definitions, and is expressed in the same modelling language as them. With this approach, since they are both high-level conceptual data models, semantic mismatches may occur between the CDM and a source data model, and there may be a loss of information between them. Moreover, if a data source schema changes, it is not straightforward to evolve the view definitions of the warehouse schema.

Lakshmanan *et al* [LSS93, LSS99, LSS01] argue that a uniform framework for schema integration and schema evolution is both desirable and possible, and this is possible with AutoMed also as we discuss in this chapter. They define a higher-order logic language, SchemaSQL, which handles data integration and schema evolution in relational multi-database systems. In contrast, our approach uses a simple set of schema transformation primitives, augmented with a functional query language, both of which are uniformly applicable to multiple data models. Other previous work on schema evolution [ALP91, Bel96, Ben99, BSH99] has also presented approaches in terms of just one data model.

In contrast to the CDM approach, AutoMed's data source wrappers translate each data source schema into its equivalent AutoMed representation, without loss of information. In Chapter 3 we discussed how AutoMed metadata can be used to express the schemas and the cleansing, transformation and integration processes in heterogeneous data warehouse environments, supporting both schema heterogeneity and model heterogeneity. It is clearly advantageous to be able to reuse this kind of metadata if a schema evolves. In this chapter we show how this can be achieved.

Earlier work [MP02] has shown how the AutoMed framework readily supports schema evolution in *virtual* data integration scenarios. This chapter addresses the problem of schema evolution in *materialised* data integration scenarios, including both evolution of a source schema and of the warehouse schema, and also the impact on any data marts derived from the warehouse. This scenario is more complex than with virtual data integration, since both schemas and materialised data may be affected by an evolution.

4.2 A Data Integration Scenario and Example

Figure 4.1 shows a data integration scenario in AutoMed.

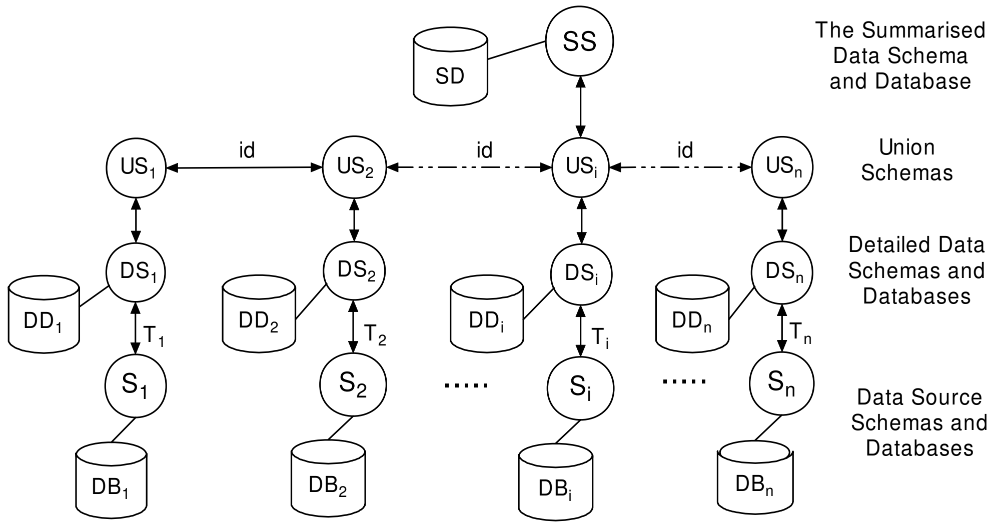

SD
SS
The Summarised Data Schema and Database
US1
id
US2
id
USi
id
USn
Union Schemas
DS1
DD1
T1
S1
DB1
DS2
DD2
T2
S2
DB2
DSi
DDi
Ti
Si
DBi
DSn
DDn
Tn
Sn
DBn
Detailed Data Schemas and Databases
Data Source Schemas and Databases

Figure 4.1: Data Integration Scenario

In this data integration scenario, each data source $\texttt{DB}_i$ is described by a data source schema $\texttt{S}_i$. Each $\texttt{S}_i$ is first conformed into a detailed data schema $\texttt{DS}_i$ (which may or may not be expressed in the same modelling language as $\texttt{S}_i$) by means of a transformation pathway T_i. The process of single-source data cleansing can be encapsulated in this transformation pathway. There may be information within the summarised data schema which is not semantically derivable from $\texttt{S}_i$, and this is asserted by the pathway from $\texttt{DS}_i$ to the 'union-schema' $\texttt{US}_i$ which consists of the necessary extend transformations[1].

All the union schemas $\texttt{US}_1, \ldots, \texttt{US}_n$ are syntactically identical and this is asserted by creating a sequence of id transformations between each pair $\texttt{US}_i$ and

[1]If there are none, then this pathway is empty and $\texttt{CS}_i$ and $\texttt{DS}_i$ are the same schema

$\mathtt{US}_{i+1}$, of the form id $\mathtt{US}_i:\mathtt{c}$ $\mathtt{US}_{i+1}:\mathtt{c}$ for each schema construct c. An id transformation signifies the semantic equivalence of syntactically identical constructs in different schemas. The transformation pathways containing these id transformations can be automatically generated by the AutoMed software. An arbitrary one of the $\mathtt{US}_i$ can then be selected for further transformation into the summarised schema SS. The extent of each construct c in a union schema $\mathtt{US}_i$ is equal to the bag-union of the extent of c in all union schemas $\mathtt{US}_1, \ldots, \mathtt{US}_n$. That is, id is interpreted as bag-union by AutoMed's view generation functionality. The processes of multi-source data cleansing, integrating and summarising can be handled over the pathway from $\mathtt{US}_i$ to SS.

We assume that all the source, detailed and summarised schemas are materialised in the databases $\mathtt{DB}_i$, $\mathtt{DD}_i$ and SD while all union schemas $\mathtt{US}_i$ are virtual. Figure 4.2 gives a concrete example of this data integration scenario.

The transformation pathway T_1 below transforms the schema $\mathtt{S}_1$ into $\mathtt{DS}_1$ by first creating Rel construct ⟨⟨MAtab⟩⟩ and its attributes ⟨⟨MAtab, Dept⟩⟩, ⟨⟨MAtab, CID⟩⟩, ⟨⟨MAtab, SID⟩⟩ and ⟨⟨MAtab, Mark⟩⟩ using add transformations, and then using delete transformations to delete the schema constructs of $\mathtt{S}_1$.

```
T1 :     S1 → DS1
addRel   ⟨⟨MAtab⟩⟩           [{'MA','MAC01',x}|x←⟨⟨MAC01⟩⟩]
                             ++[{'MA','MAC02',x}|x←⟨⟨MAC02⟩⟩]
                             ++[{'MA','MAC03',x}|x←⟨⟨MAC03⟩⟩];
addAtt   ⟨⟨MAtab, Dept⟩⟩     [{k1,k2,k3,k1}|{k1,k2,k3}←⟨⟨MAtab⟩⟩];
addAtt   ⟨⟨MAtab, CID⟩⟩      [{k1,k2,k3,k2}|{k1,k2,k3}←⟨⟨MAtab⟩⟩];
addAtt   ⟨⟨MAtab, SID⟩⟩      [{k1,k2,k3,k3}|{k1,k2,k3}←⟨⟨MAtab⟩⟩];
addAtt   ⟨⟨MAtab, Mark⟩⟩     [{'MA','MAC01',k,x}|{k,x}←⟨⟨MAC01, Mark⟩⟩]
                             ++[{'MA','MAC02',k,x}|{k,x}←⟨⟨MAC02, Mark⟩⟩]
                             ++[{'MA','MAC03',k,x}|{k,x}←⟨⟨MAC03, Mark⟩⟩];
```

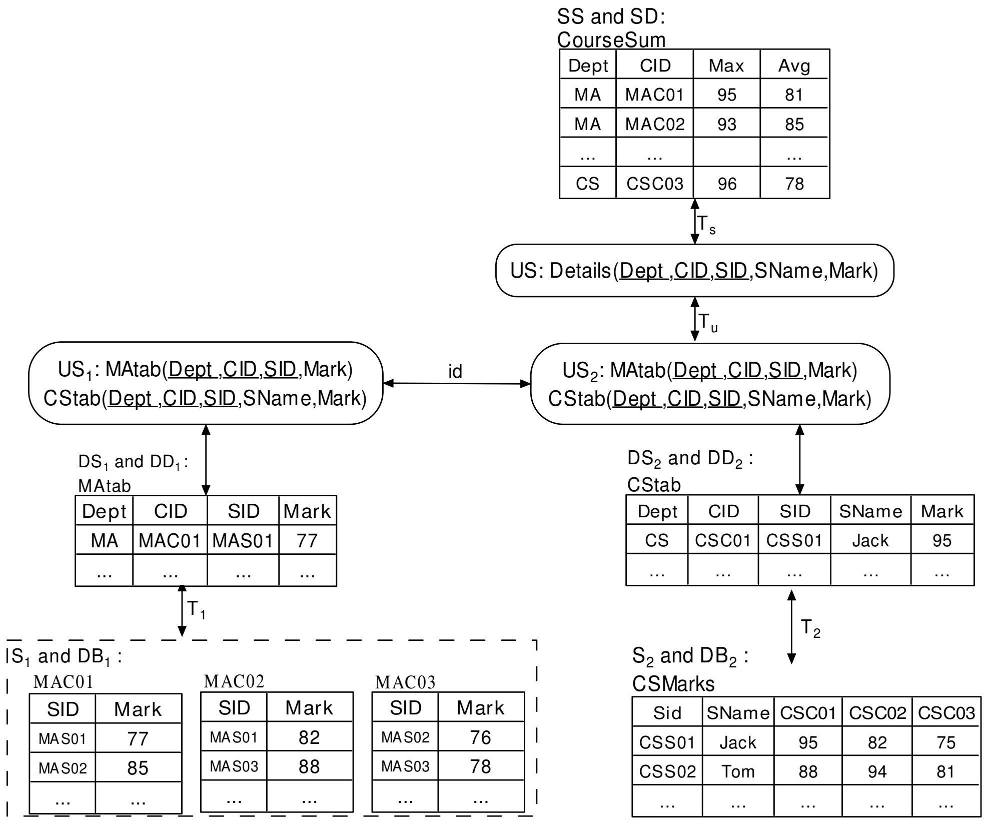


Figure 4.2: Example of Data Integration

delAtt	⟨⟨MAC01, Mark⟩⟩	`[{k3,x}|{k1,k2,k3,x}←⟨⟨MAtab, Mark⟩⟩;k2='MAC01']`;
delAtt	⟨⟨MAC01, SID⟩⟩	`[{k3,x}|{k1,k2,k3,x}←⟨⟨MAtab, SID⟩⟩;k2='MAC01']`;
delRel	⟨⟨MAC01⟩⟩	`[{k3}|{k1,k2,k3}←⟨⟨MAtab⟩⟩;k2='MAC01']`;
delAtt	⟨⟨MAC02, Mark⟩⟩	`[{k3,x}|{k1,k2,k3,x}←⟨⟨MAtab, Mark⟩⟩;k2='MAC02']`;
delAtt	⟨⟨MAC02, SID⟩⟩	`[{k3,x}|{k1,k2,k3,x}←⟨⟨MAtab, SID⟩⟩;k2='MAC02']`;
delRel	⟨⟨MAC02⟩⟩	`[{k3}|{k1,k2,k3}←⟨⟨MAtab⟩⟩;k2='MAC02']`;
delAtt	⟨⟨MAC03, Mark⟩⟩	`[{k3,x}|{k1,k2,k3,x}←⟨⟨MAtab, Mark⟩⟩;k2='MAC03']`;
delAtt	⟨⟨MAC03, SID⟩⟩	`[{k3,x}|{k1,k2,k3,x}←⟨⟨MAtab, SID⟩⟩;k2='MAC03']`;
delRel	⟨⟨MAC03⟩⟩	`[{k3}|{k1,k2,k3}←⟨⟨MAtab⟩⟩;k2='MAC03']`;

The transformation pathway T_2 below transforms schema $\mathtt{S}_2$ into $\mathtt{DS}_2$:

T_2 :	$\mathtt{S}_2 \rightarrow \mathtt{DS}_2$	
addRel	⟨⟨CStab⟩⟩	`[{'CS',x,y}|x← ['CSC01','CSC02','CSC03'];` `y`← ⟨⟨CSMarks⟩⟩];
addAtt	⟨⟨CStab, Dept⟩⟩	`[{k1,k2,k3,k1}|{k1,k2,k3}`← ⟨⟨CStab⟩⟩];
addAtt	⟨⟨CStab, CID⟩⟩	`[{k1,k2,k3,k2}|{k1,k2,k3}`← ⟨⟨CStab⟩⟩];
addAtt	⟨⟨CStab, SID⟩⟩	`[{k1,k2,k3,k3}|{k1,k2,k3}`← ⟨⟨CStab⟩⟩];
addAtt	⟨⟨CStab, SName⟩⟩	`[{'CS',x,k,s}|x← ['CSC01','CSC02','CSC03'];` `{k,s}`← ⟨⟨CSMarks, SName⟩⟩];
addAtt	⟨⟨CStab, Mark⟩⟩	`[{'CS','CSC01',k,x}|{k,x}`← ⟨⟨CSMarks, CSC01⟩⟩] `++[{'CS','CSC02',k,x}|{k,x}`← ⟨⟨CSMarks, CSC02⟩⟩] `++[{'CS','CSC03',k,x}|{k,x}`← ⟨⟨CSMarks, CSC03⟩⟩];
delAtt	⟨⟨CSMarks, CSC03⟩⟩	`[{s,m}|{d,c,s,m}` ← ⟨⟨CStab, Mark⟩⟩; `c='CSC03'`];
delAtt	⟨⟨CSMarks, CSC02⟩⟩	`[{s,m}|{d,c,s,m}` ← ⟨⟨CStab, Mark⟩⟩; `c='CSC02'`];

delAtt	⟨⟨CSMarks, CSC01⟩⟩	`[{s,m}|{d,c,s,m}` ← ⟨⟨CStab, Mark⟩⟩; `c='CSC01'`];
delAtt	⟨⟨CSMarks, SName⟩⟩	`distinct [{s,n}|{d,c,s,n}` ← ⟨⟨CStab, SName⟩⟩];
delAtt	⟨⟨CSMarks, Sid⟩⟩	`distinct [{s,i}|{d,c,s,i}` ← ⟨⟨CStab, SID⟩⟩];
delRel	⟨⟨CSMarks⟩⟩	`distinct [s|{d,c,s}` ← ⟨⟨CStab⟩⟩];

Since $\mathtt{US}_1$ contains schema constructs of relation `CStab` which do not appear in $\mathtt{DS}_1$, the transformation pathway $\mathtt{DS}_1 \rightarrow \mathtt{US}_1$ contains extend transformations extending these constructs into $\mathtt{DS}_1$:

extendAtt	⟨⟨CStab, Mark⟩⟩	`Void;`
extendAtt	⟨⟨CStab, SName⟩⟩	`Void;`
extendAtt	⟨⟨CStab, SID⟩⟩	`Void;`
extendAtt	⟨⟨CStab, CID⟩⟩	`Void;`
extendAtt	⟨⟨CStab, Dept⟩⟩	`Void;`
extendRel	⟨⟨CStab⟩⟩	`Void;`

Similarly, the transformation pathway $DS_2 \to US_2$ contains extend transformations extending the schema constructs of relation MAtab into DS_2.

A sequence of id transformations is created between US_1 and US_2, and US_2 is selected for further transformation. In this example, we transform US_2 into US, which integrates the two relations MAtab and CStab into a relation Details, using the following transformation pathway T_u (note that US_1, US_2 and US are all virtual schemas):

```
T_u :     US_2 → US
addRel   ⟨⟨Details⟩⟩           ⟨⟨MAtab⟩⟩ ++ ⟨⟨CStab⟩⟩;
addAtt   ⟨⟨Details, Dept⟩⟩     ⟨⟨MAtab, Dept⟩⟩ ++ ⟨⟨CStab, Dept⟩⟩;
addAtt   ⟨⟨Details, CID⟩⟩      ⟨⟨MAtab, CID⟩⟩ ++ ⟨⟨CStab, CID⟩⟩;
addAtt   ⟨⟨Details, SID⟩⟩      ⟨⟨MAtab, SID⟩⟩ ++ ⟨⟨CStab, SID⟩⟩;
addAtt   ⟨⟨Details, SName⟩⟩    ⟨⟨MAtab, SName⟩⟩ ++ ⟨⟨CStab, SName⟩⟩;
addAtt   ⟨⟨Details, Mark⟩⟩     ⟨⟨MAtab, Mark⟩⟩ ++ ⟨⟨CStab, Mark⟩⟩;
delAtt   ⟨⟨MAtab, Mark⟩⟩      [{d,c,s,m}|{d,c,s,m} ← ⟨⟨Details, Mark⟩⟩; d='MA'];
delAtt   ⟨⟨MAtab, SName⟩⟩     [{d,c,s,n}|{d,c,s,n} ← ⟨⟨Details, SName⟩⟩; d='MA'];
delAtt   ⟨⟨MAtab, SID⟩⟩       [{d,c,s,i}|{d,c,s,i} ← ⟨⟨Details, SID⟩⟩; d='MA'];
delAtt   ⟨⟨MAtab, CID⟩⟩       [{d,c,s,i}|{d,c,s,i} ← ⟨⟨Details, CID⟩⟩; d='MA'];
delAtt   ⟨⟨MAtab, Dept⟩⟩      [{d,c,s,i}|{d,c,s,i} ← ⟨⟨Details, Dept⟩⟩; d='MA'];
delRel   ⟨⟨MAtab⟩⟩            [{d,c,s}|{d,c,s} ← ⟨⟨Details, Mark⟩⟩; d='MA'];
delAtt   ⟨⟨CStab, Mark⟩⟩      [{d,c,s,m}|{d,c,s,m} ← ⟨⟨Details, Mark⟩⟩; d='CS'];
delAtt   ⟨⟨CStab, SName⟩⟩     [{d,c,s,n}|{d,c,s,n} ← ⟨⟨Details, SName⟩⟩; d='CS'];
delAtt   ⟨⟨MAtab, SID⟩⟩       [{d,c,s,i}|{d,c,s,i} ← ⟨⟨Details, SID⟩⟩; d='CS'];
delAtt   ⟨⟨CStab, CID⟩⟩       [{d,c,s,i}|{d,c,s,i} ← ⟨⟨Details, CID⟩⟩; d='CS'];
delAtt   ⟨⟨CStab, Dept⟩⟩      [{d,c,s,i}|{d,c,s,i} ← ⟨⟨Details, Dept⟩⟩; d='CS'];
delRel   ⟨⟨CStab⟩⟩            [{d,c,s}|{d,c,s} ← ⟨⟨Details, Mark⟩⟩; d='CS'];
```

The transformation pathway T_s finally transforms schema US into SS, where contract transformations are used to contract the schema constructs in US that cannot be recovered from SS.

```
Ts :          US → SS
addRel        ⟪CourseSum⟫          distinct [{k1,k2}|{k1,k2,k3}←⟪Details⟫];
addAtt        ⟪CourseSum, Dept⟫    [{k1,k2,k1}|{k1,k2}←⟪CourseSum⟫];
addAtt        ⟪CourseSum, CID⟫     [{k1,k2,k2}|{k1,k2}←⟪CourseSum⟫];
addAtt        ⟪CourseSum, Max⟫     [{x,y,z}|{{x,y},z}←(gc max [{{k1,k2},x}|
                                                 {k1,k2,k3,x}←⟪Details, Mark⟫])];
addAtt        ⟪CourseSum, Avg⟫     [{x,y,z}|{{x,y},z}←(gc avg [{{k1,k2},x}|
                                                 {k1,k2,k3,x}←⟪Details, Mark⟫])];
contractAtt   ⟪Details, Mark⟫;
contractAtt   ⟪Details, SName⟫;
contractAtt   ⟪Details, CName⟫;
contractAtt   ⟪Details, SID⟫;
contractAtt   ⟪Details, CID⟫;
contractAtt   ⟪Details, Dept⟫;
contractRel   ⟪Details⟫;
```

4.3 Expressing Schema and Data Model Evolution

In a heterogeneous data warehousing environment, it is possible for either a data source schema or the integrated database schema to evolve. This schema evolution may be a change in the schema, or a change in the data model in which the schema is expressed, or both. AutoMed transformations can be used to express the schema evolution in all three cases:

(a) Consider first a schema S expressed in a modelling language $\mathcal{M}$. We can express the evolution of S to S^{new}, also expressed in $\mathcal{M}$, as a series of primitive transformations that rename, add, extend, delete or contract constructs of $\mathcal{M}$. For example, suppose that the relational schema $\mathtt{S}_1$ in the above example evolves so its three tables become a single table with an extra column for the course ID. This evolution is captured by a pathway which is identical to the pathway $\mathtt{S}_1 \rightarrow \mathtt{DS}_1$ given above.

This kind of transformation that captures well-known equivalences between schemas [LNE89, MP98] can be defined in AutoMed by means of a parametrised transformation *template* which is both schema- and data-independent. When invoked with specific schema constructs and their extents, a template generates the appropriate sequence of primitive transformations within the Schemas & Transformations Repository.

(b) Consider now a schema S expressed in a modelling language $\mathcal{M}$ which evolves into an equivalent schema S^{new} expressed in a modelling language $\mathcal{M}^{new}$. We can express this translation by a series of add steps that define the constructs of S^{new} in $\mathcal{M}^{new}$ in terms of the constructs of S in $\mathcal{M}$. At this stage, we have an intermediate schema that contains the constructs of both S and S^{new}. We then specify a series of delete steps that remove the constructs of $\mathcal{M}$ (the queries within these transformations indicate that these are now redundant constructs since they can be derived from the new constructs).

The example in Section 3.3.1 shows how evolutions between schemas expressed in different modelling languages can be captured by transformation pathways. Again, generic inter-model translations between one data model and another can be defined in AutoMed by means of transformation templates.

(c) Considering finally to an evolution which is both a change in the schema and in the data model, this can be expressed by a combination of (a) and

(b) above: either (a) followed by (b), or (b) followed by (a), or indeed by interleaving the two processes.

4.4 Handling Schema Evolution

We now consider how the integration network illustrated in Figure 4.1 is evolvable in the face of evolution of a data source schema or the summarised data schema. We have seen in the previous section how AutoMed transformations can be used to express the schema evolution if either the schema or the data model changes, or both. We can therefore treat schema and data model change in a uniform way for the purposes of handling schema evolution: both are expressed as a sequence of AutoMed primitive transformations, in the first case staying within the original data model, and in the second case transforming the original schema in the original data model into a new schema in a new data model.

In this section we describe the actions that are taken in order to evolve the integration network of Figure 4.1 if the summarised data schema SS evolves (Section 4.4.1) or if a data source schema $\mathtt{S}_i$ evolves (Section 4.4.2). Given an evolution pathway from a schema S to a schema S^{new}, in both cases each successive primitive transformation within the pathway $S \rightarrow S^{new}$ is treated one at a time. Thus, we describe in sections 4.4.1 and 4.4.2 the actions that are taken if $S \rightarrow S^{new}$ consists of just one primitive transformation. If $S \rightarrow S^{new}$ is a composite transformation, then it is handled as a sequence of primitive transformations.

Our discussion below assumes that the primitive transformation being handled is adding, removing or renaming a construct of S that has an underlying data extent.

4.4.1 Evolution of the Summarised Data Schema

Suppose the summarised data schema SS evolves by means of a primitive transformation t into $\texttt{SS}^{new}$. This is expressed by the step t being appended to the pathway T_u of Figure 4.1. The new summarised data schema is $\texttt{SS}^{new}$ and its associated extension is $\texttt{SD}^{new}$. SS is now an intermediate schema in the extended pathway $T_u; t$ and it no longer has an extension associated with it. t may be a rename, add, extend, delete or contract transformation. The following actions are taken in each case:

1. If t is renameT(c,c'), then there is nothing further to do. SS is semantically equivalent to $\texttt{SS}^{new}$ and $\texttt{SD}^{new}$ is identical to SD except that the extent of c in SD is now the extent of c' in $\texttt{SD}^{new}$.

2. If t is addT(c,q), then there is nothing further to do at the schema level. SS is semantically equivalent to $\texttt{SS}^{new}$. However, the new construct c in $\texttt{SD}^{new}$ must now be populated, and this is achieved by evaluating the query q over SD.

3. If t is extendT(c)[2] then the new construct c in $\texttt{SD}^{new}$ is populated by an empty extent. This new construct may subsequently be populated by an expansion in a data source (see Section 4.4.2).

4. If t is deleteT(c,q) or contractT(c), then the extent of c must be removed from SD in order to create $\texttt{SD}^{new}$ (it is assumed that this a legal deletion/contraction, e.g if we wanted to delete/contract a table from a relational schema, then first the constraints and then the columns would be

[2]For this chapter, we assume that extend and contract transformations have lower-bound queries Void and upper-bound queries Any, and we denote them as extendT(c) and contractT(c). We leave as further work handling schema evolution for more general extend and contract transformations.

deleted/contracted and lastly the table itself; such syntactic correctness of transformation pathways is automatically verified by AutoMed). It may now be possible to simplify the transformation network, in that if T_u contains a matching transformation addT(c,q) or extendT(c), then both this and the new transformation t can be removed from the pathway US $\rightarrow$ SS^{new}. This is purely an optimization — it does not change the meaning of a pathway, nor its effect on view generation and query/data translation. We refer the reader to [Ton03] for details of the algorithms that simplify AutoMed transformation pathways.

In cases 2 and 3 above, the new construct c will automatically be propagated into the schema DMS of any data mart derived from SS. To prevent this, a transformation contractT(c) can be prefixed to the pathway SS $\rightarrow$ DMS. Alternatively, the new construct c can be propagated to DMS if so desired, and materialised there. In cases 1 and 4 above, the change in SS and SD may impact on the data marts derived from SS, and we discuss this in Section 4.4.3.

4.4.2 Evolution of a Data Source Schema

Suppose a data source schema $\mathtt{S}_i$ evolves by means of a primitive transformation t into $\mathtt{S}_i^{new}$. As discussed in Chapter 3, there is automatically available a reverse transformation t^{-1} from $\mathtt{S}_i^{new}$ to $\mathtt{S}_i$ and hence a pathway $t^{-1}; T_i$ from $\mathtt{S}_i^{new}$ to $\mathtt{DS}_i$. The new data source schema is $\mathtt{S}_i^{new}$ and its associated extension is $\mathtt{DB}_i^{new}$. $\mathtt{S}_i$ is now just an intermediate schema in the extended pathway $t^{-1}; T_i$ and it no longer has an associated extension.

t may be a rename, add, delete, extend or contract transformation. In 1–5 below we see what further actions are taken in each case for evolving the integration network and the downstream materialised data as necessary.

We first introduce some necessary terminology: If p is a pathway $S \to S'$ and c is a construct in S, we denote by $\mathsf{descendants}(\mathsf{c}, p)$ the constructs of S' which are directly or indirectly dependent on c, either because c itself appears in S' or because a construct c' of S' is created by a transformation $\mathsf{addT}(\mathsf{c}', \mathsf{q})$ within p where the query q directly or indirectly references c. The set $\mathsf{descendants}(\mathsf{c}, p)$ can be straight-forwardly computed by traversing p and inspecting the query associated with each add transformation within in.

1. If t is renameT(c,c'), then schema $\mathtt{S}_i^{new}$ is semantically equivalent to $\mathtt{S}_i$. The new transformation pathway T_i^{new} : $\mathtt{S}_i^{new} \to \mathtt{DS}_i$ is $t^{-1}; T_i$ = renameT(c',c); T_i. The new source database $\mathtt{DB}_i^{new}$ is identical to $\mathtt{DB}_i$ except that the extent of c in $\mathtt{DB}_i$ is now the extent of c' in $\mathtt{DB}_i^{new}$.

2. If t is addT(c,q), then $\mathtt{S}_i$ has evolved to contain a new construct c whose extent is equivalent to the expression q over the other constructs of $\mathtt{S}_i$. The new transformation pathway T_i^{new} : $\mathtt{S}_i^{new} \to \mathtt{DS}_i$ is $t^{-1}; T_i$ = deleteT(c,q); T_i.

3. If t is deleteT(c,q), this means that $\mathtt{S}_i$ has evolved to not include a construct c whose extent is derivable from the expression q over the other constructs of $\mathtt{S}_i$, and the new source database $\mathtt{DB}_i^{new}$ no longer contains an extent for c. The new transformation pathway T_i^{new} : $\mathtt{S}_i^{new} \to \mathtt{DS}_i$ is $t^{-1}; T_i$ = addT(c,q); T_i.

In the above three cases, schema $\mathtt{S}_i^{new}$ is semantically equivalent to $\mathtt{S}_i$, and nothing further needs to be done to any of the transformation pathways, schemas or databases $\mathtt{DD}_1$, ..., $\mathtt{DD}_n$ and SD. This may not be the case if t is a contract or extend transformation, which we consider next.

4. If t is extendT(c), then there will be a new construct available from $\mathtt{S}_i^{new}$ that was not available before. That is, $\mathtt{S}_i$ has evolved to contain the new construct c whose extent is not derivable from the other constructs of $\mathtt{S}_i$.

If we left the transformation pathway T_i as it is, this would result in a pathway $T_i^{new} = \mathsf{contractT(c)}; T_i$ from $\mathtt{S}_i^{new}$ to $\mathtt{DS}_i$, which would immediately drop the new construct $\mathtt{c}$ from the integration network. That is, T_i^{new} is consistent but it does not utilize the new data.

However, recall that we said earlier that we assume no contract steps in the pathways from the data schemas to their union schemas, and that all the data in $\mathtt{S}_i$ should be available to the integration network. In order to achieve this, there are four cases to consider if t is extendT(c):

(4.a) $\mathtt{c}$ appears in $\mathtt{US}_i$ and has the same semantics as the newly added $\mathtt{c}$ in $\mathtt{S}_i^{new}$.

Since $\mathtt{c}$ cannot be derived from the original $\mathtt{S}_i$, there must be a transformation `extendT(c)`, in $\mathtt{DS}_i \rightarrow \mathtt{US}_i$.

We remove from T_i^{new} the new contractT(c) step and this matching `extendT(c)` step. This propagates $\mathtt{c}$ into $\mathtt{DS}_i$, and we populate its extent in the materialised database $\mathtt{DD}_i$ by replicating its extent from $\mathtt{DB}_i^{new}$.

(4.b) $\mathtt{c}$ does not appear in $\mathtt{US}_i$ but it can be derived from $\mathtt{US}_i$ by means of some transformation T.

In this case, we remove from T_i^{new} the first contractT(c) step, so that $\mathtt{c}$ is now present in $\mathtt{DS}_i$ and in $\mathtt{US}_i$. We populate the extent of $\mathtt{c}$ in $\mathtt{DD}_i$ by replicating its extent from $\mathtt{DB}_i^{new}$.

To repair the other pathways $T_j : \mathtt{S}_j \rightarrow \mathtt{DS}_j$ and schemas $\mathtt{US}_j$ for $j \neq i$, we append T to the end of each T_j. As a result, the new construct $\mathtt{c}$ now appears in all the union schemas. To add the extent of this new construct to each materialised database $\mathtt{DD}_j$ for $j \neq i$, we compute it from the extents of the other constructs in $\mathtt{DS}_j$ using the queries within successive add steps in T.

We finally append the necessary new id steps between pairs of union schemas to assert the semantic equivalence of the construct c within them.

(4.c) c does not appear in $\texttt{US}_i$ and cannot be derived from $\texttt{US}_i$.

In this case, we again remove from T_i^{new} the first contractT(c) step so that c is now present in schema $\texttt{DS}_i$.

To repair the other pathways $T_j : \texttt{S}_j \rightarrow \texttt{DS}_j$ and schemas $\texttt{US}_j$ for $j \neq i$, we append an extendT(c) step to the end of each T_j. As a result, the new construct c now appears in all the conformed schemas $\texttt{DS}_1, \ldots, \texttt{DS}_n$.

The construct c may need further translation into the data model of the union schemas and this is done by appending the necessary sequence, T, of add/delete/rename steps to all the pathways $\texttt{S}_1 \rightarrow \texttt{DS}_1, \ldots, \texttt{S}_n \rightarrow \texttt{DS}_n$.

We compute the extent of c within the database $\texttt{DD}_i$ from its extent within $\texttt{DB}_i^{new}$ using the queries within successive add steps in T.

We finally append the necessary new id steps between pairs of union schemas to assert the semantic equivalence of the new construct(s) within them.

(4.d) c appears in $\texttt{US}_i$ but has different semantics to the newly added c in $\texttt{S}_i^{new}$.

In this case, we rename c in $\texttt{S}_i^{new}$ to a new construct c'. The situation reverts to adding a new construct c' to $\texttt{S}_i^{new}$, and one of (4.a)-(4.c) above applies.

We note that determining whether c can or cannot be derived from the existing constructs of the union schemas in (4.a)–(4.d) above requires domain or expert human knowledge. Thereafter, the remaining actions are fully automatic.

In cases (4.a) and (4.b), there is new data added to one or more of the conformed databases which needs to be propagated to SD. This is done by computing descendants$(\texttt{c}, T_u)$ and using the algebraic equivalences of IQL syntax given in Chapter 3 to propagate changes in the extent of `c` to each of its descendant constructs `dc` in `SS`. Using these equivalences, we can in most cases incrementally recompute the extent of `dc`. If at any stage in T_u there is a transformation $\mathsf{addT}(\mathsf{c}', \mathsf{q})$ where no equivalence can be applied, then we have to recompute the whole extent of `c'`.

In cases (4.b) and (4.c), there is a new schema construct `c` appearing in the $\mathtt{US}_i$. This construct will automatically appear in the schema `SS`. If this is not desired, a transformation $\mathsf{contractT}(\mathsf{c})$ can be prefixed to T_u.

5. If t is $\mathsf{contractT}(\mathsf{c})$, then the construct `c` in $\mathtt{S}_i$ will no longer be available from $\mathtt{S}_i^{new}$. That is, $\mathtt{S}_i$ has evolved so as to not include a construct `c` whose extent is not derivable from the other constructs of $\mathtt{S}_i$. The new source database $\mathtt{DB}_i^{new}$ no longer contains an extent for `c`.

   The new transformation pathway $T_i^{new} : \mathtt{S}_i^{new} \rightarrow \mathtt{DS}_i$ is $t^{-1}; T_i = \mathsf{extendT}(\mathsf{c}); T_i$. Since the extent of `c` is now Void, the materialised data in $\mathtt{DD}_i$ and `SD` must be modified so as to remove any data derived from the old extent of `c`.

   In order to repair $\mathtt{DD}_i$, we compute $\mathsf{descendants}(\mathsf{c}, \mathtt{S}_i \rightarrow \mathtt{DS}_i)$. For each construct `uc` in $\mathsf{descendants}(\mathsf{c}, \mathtt{S}_i \rightarrow \mathtt{DS}_i)$, we compute its new extent and replace its old extent in $\mathtt{DD}_i$ by the new extent. Again, the algebraic properties of IQL queries discussed in Chapter 3 can be used to propagate the new Void extent of construct `c` in $\mathtt{S}_i^{new}$ to each of its descendant constructs `uc` in $\mathtt{DS}_i$. Using these equivalences, we can in most cases incrementally recompute the extent of `uc` as we traverse the pathway T_i.

   In order to repair `SD`, we similarly propagate changes in the extent of each `uc` along the pathway T_u.

   Finally, it may also be necessary to amend the transformation pathways if there are one or more constructs in SD which now will always have an empty extent as a result of this contraction of $\mathtt{S}_i$. For any construct uc in US whose extent has become empty, we examine all pathways $T_1, \ldots, T_n$. If all these pathways contain an extendT(uc) transformation, or if using the equivalences of IQL syntax in Chapter 3 we can deduce from them that the extent of uc will always be empty, then we can suffix a contractT(dc) step to T_u for every dc in descendants(uc, T_u), and then handle this case as paragraph 4 in Section 4.4.1.

4.4.3 Evolution of Downstream Data Marts

We have discussed how evolutions to the summarised data schema or to a source schema are handled. One remaining question is how to handle the impact of a change to the data warehouse schema, and possibly its data, on any data marts that have been derived from it.

In Chapter 3 we discuss how it is possible to express the derivation of a data marts from a data warehouse by means of an AutoMed transformation pathway. Such a pathway DWS $\rightarrow$ DMS expresses the relationship of a data mart schema DMS to the warehouse schema DWS. As such, this scenario can be regarded as a special case of the general integration scenario of Figure 4.1, where SS now plays the role of the single source schema, databases $\mathtt{DD}_1, \ldots, \mathtt{DD}_n$ and SD collectively play the role of the data associated with this source schema and DMS plays the role of the summarised data schema. Therefore, the same techniques as discussed in sections 4.4.1 and 4.4.2 can be applied.

4.5 Discussion

In this chapter we have described how the AutoMed heterogeneous data integration toolkit can be used to handle the problem of schema evolution in heterogeneous data warehousing environments so that the previous transformation, integration and data materialisation effort can be reused. We have discussed handling evolution of a source schema or the warehouse schema, and also the impact on any downstream data marts derived from the data warehouse. Our techniques are mainly automatic, except for the aspects that require domain or expert human knowledge regarding the semantics of new schema constructs.

We have shown how AutoMed transformations can be used to express schema evolution within the same data model, or a change in the data model, or both, whereas other schema evolution literature has focussed on just one data model. Schema evolution within the relational data model has been discussed in previous work such as [LSS93, LSS99, Mil98]. The approach in [Mil98] uses a first-order schema in which all values in a schema of interest to a user are modelled as data, and other schemas can be expressed as a query over this first-order schema. The approach in [LSS99] uses the notation of a *flat scheme*, and gives four operators Unite, Fold, Unfold and Split to perform relational schema evolution using the SchemaSQL language. In contrast, with AutoMed the process of schema evolution is expressed using a simple set of primitive schema transformations augmented with a functional query language, both of which are applicable to multiple data models.

Our approach is complementary to work on mapping composition, e.g. [VMP03, MH03, FKP04], in that in our case the new mappings are a composition of the original transformation pathway and the transformation pathway which expresses the schema evolution. Thus, the new mappings are, by definition, correct. There are two aspects to our approach:

(i) handling the transformation pathways and

(ii) handling the queries within them.

In this chapter we have in particular assumed that the queries are expressed in IQL. However, the AutoMed toolkit allows any query language syntax to be used within primitive transformations, and therefore this aspect of our approach could be extended to other query languages.

Materialised data warehouse views need to be maintained when the data sources change, and much previous work has addressed this problem at the data level. However, as we have discussed in this chapter, materialised data warehouse views may also need to be modified if there is an evolution of a data source schema. Incremental maintenance of schema-restructuring views within the relational data model is discussed in [KR02], whereas our approach can handle this problem in a heterogeneous data warehousing environment with multiple data models and changes in data models. In chapter 7, we will discuss how AutoMed transformation pathways can also be used for incrementally maintaining materialised views at the data level.

Chapter 5

Using Materialised AutoMed Transformation Pathways for Data Lineage Tracing

The data lineage tracing problem is to find the *derivation* of the given *tracing data* in the global database. The derivation, called the *lineage data*, is a collection of data items in the data sources which produces the given tracing data. The tracing data consists of data item(s) in the global database, which may be a single tuple, called the *tracing tuple*, or a set of tuples, called the *tracing tuples*.

In this chapter, we will give the definitions of data lineage in the context of AutoMed, and develop a set of algorithms which use materialised AutoMed schema transformation pathways for tracing data lineage. By materialised, we mean that all intermediate schema constructs created in the schema transformations are materialised, *i.e.* have an extent associated with them.

We consider a subset of the full IQL query language which incorporates the major relational and aggregation operators on collections. We call this subset IQL^c and its syntax is as follows, where $\mathtt{E}$, $\mathtt{E}_1 \ldots, \mathtt{E}_n$ denote collection-valued IQL^c

queries; $\mathtt{e}_1, ..., \mathtt{e}_n$ are constants, variables or IQL^c queries; `f` is an aggregation function (`max`, `min`, `count`, `sum`, `avg`); `p`, `p1`, `p2` denote patterns; and $\mathtt{Q}_1 ... \mathtt{Q}_n$ are qualifiers which may be generators or filters. Filters in IQL^c are limited to boolean-valued expressions containing only variables, constants and comparison operators and expressions of the form `member E x` and `not (member E x)`.

1. $[\mathtt{e}_1, \mathtt{e}_2, ..., \mathtt{e}_n]$
2. `group E`
3. `sort E`
4. `distinct E`
5. `f E`
6. `gc f E`
7. $\mathtt{E}_1 ++ \mathtt{E}_2 ++ \ldots ++ \mathtt{E}_n$
8. $\mathtt{E}_1 -- \mathtt{E}_2$
9. $[\mathtt{p}|\mathtt{Q}_1; \ldots; \mathtt{Q}_n]$
10. `map (lambda p1.p2) E`

This subset of IQL can express the common algebraic operations on collections. In particular, let us consider *select*(σ), *projection*(π), *join*($\bowtie$) and *aggregation*(α) (*union* and *difference* are directly supported in IQL^c via the ++ and −− operators). The general form of a select-project-join (SPJ) expression is $\pi_{\mathtt{A}}(\sigma_{\mathtt{C}}(\mathtt{E}_1 \bowtie ... \bowtie \mathtt{E}_n))$ and this can be expressed in IQL^c as a comprehension of the form $[\mathtt{A}|\overline{\mathtt{x}_1} \leftarrow \mathtt{E}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{E}_n; \mathtt{C}]$. The algebraic operator α applies an aggregation function to a collection and this functionality is captured in IQL^c by the `gc` operator. For example, supposing `D` is a collection of three-tuples and has scheme `D(A1,A2,A3)`, the expression $\alpha_{\mathtt{A2},\mathtt{f(A3)}}(\mathtt{D})$ is expressed in IQL^c as `gc f (map (lambda {x1,x2,x3}.{x2,x3}) D)`

Section 5.1 below discusses related work on data lineage tracing. Section 5.2 introduces a subset of IQL^c, *simple IQL* (SIQL), for developing our data lineage tracing formulae, and presents the rules of decomposing IQL^c queries into SIQL queries. Any IQL^c query can be encoded as a series of transformations with SIQL queries on intermediate schema constructs. Section 5.3 presents the definitions of data lineage in the context of AutoMed. Sections 5.4 and 5.5 present our approach to data lineage tracing using materialised AutoMed schema transformation pathways, including formulae and algorithms. Section 5.6 discusses how the order of traversing an IQL^c query tree to decompose it into a series of SIQL queries does not affect the result of our DLT process. Section 5.7 discusses the problem of *derivation ambiguity* in data lineage tracing, and how this problem may happen and may be avoided in our context. Finally, Section 5.8 presents a summary and discussion of this chapter.

5.1 Related Work

The problem of data lineage tracing (DLT) in data warehousing environments has been studied by Cui *et al.* in [CWW00, CW00a, CW00b, CW01, Cui01]. In particular, the fundamental definitions regarding data lineage, including *tuple derivation for an operator* and *tuple derivation for a view*, were developed in [CWW00], as were methods for derivation tracing with both *set* and *bag* semantics. Their work has addressed the derivation tracing problem and has provided the concept of *derivation set* and *derivation pool* for DLT with duplicate elements. The derivation set is the set of the tuples in the tracing data's derivation excluding any duplicate elements. The derivation pool contains all tuples in the tracing data's derivation. References [CW00a, CW00b] also introduce a way to perform data lineage tracing for data warehouse views. Several DLT algorithms are provided by selecting a set of auxiliary views to materialise in the data warehouse. However, the approach is limited to the relational data model only.

Another fundamental concept of data lineage is discussed by Buneman *et al.*, in [BKT00, BKT01], namely the difference between "why" provenance and "where" provenance. Why-provenance refers to the source data that had some influence on the existence of the integrated data. Where-provenance refers to the actual data in the sources from which the integrated data was extracted.

In our approach, both why- and where-provenance are considered, using bag semantics. We use Cui's notion of derivation-pool to define the *affect-pool* and the *origin-pool* for data lineage tracing in AutoMed — the former derives all of the source data that had some influence on the tracing data, while the latter derives the specific data in the sources from which the tracing data was extracted. In contrast, Cui's definitions and methods are limited to why-provenance.

We develop formulae for deriving the affect-pool and origin-pool of a data item in the extent of a materialised schema construct created by a single schema transformation step. Our DLT approach is to apply these formulae on each transformation step in a transformation pathway in turn, so as to obtain the lineage data in stepwise fashion. The queries within transformation steps are assumed to be IQL^c queries.

Reference [KLM$^+$97] also introduces a notion of *derivation sets* for a tuple in a materialised view defined by a single-block SQL query. This represents the set of all tuples whose insertion, deletion or modification could potentially affect the tuple in the view. But this work does not focus on how to trace the derivation sets.

Cui and Widom in [CW01] discuss the problem of tracing data lineage for general data warehousing transformations, that is, the considered operators and algebraic properties are no longer limited to relational views. However, without a framework for expressing general transformations in heterogeneous database environments, most of the algorithms in [CW01] are recalling the view definition and examining each item in the data source to decide if the item is in the data lineage of the data being traced. This can be expensive if the view definition is a complex one and enumerating all items in the data source is impractical for large data sets.

Reference [WS97] proposes a general framework for computing *fine-grained* data lineage, *i.e.* a specific derivation in the data sources, using a limited amount of information, *weak* and *verified inversion*, about the processing steps. Based on weak and verified inversion functions, which must be specified by the transformation definer, the paper defines and traces data lineage for each transformation step. However, the system cannot obtain the exact lineage data, only a number of guarantees about the lineage is provided. Further, specifying weak and verified inversion functions for each transformation step is onerous work for the data warehouse definer. Moreover, the DLT process cannot straightforwardly be reused when the data warehouse evolves. Our approach considers the problem of data lineage tracing at the tuple level and computes the exact lineage data. Moreover, AutoMed's ready support for schema evolution means that our DLT algorithms can be reapplied if schema transformation pathways evolve.

There are also other previous works relating to data lineage tracing, such as [BB99, HQGW93, FJS97], which consider *coarse-grained* lineage based on annotations on each data transformation step, and provide estimated lineage information rather than the exact data items in the data sources. Reference [BB99] presents a schema whereby each data warehouse row generated by the data warehousing transformations is tagged by an identifier for the transformation, so that the user can trace which transformation generated each data warehouse row. Reference [HQGW93] uses Petri Nets to model and capture data derivations in scientific databases, which record the derivation relationships among classes of data. Reference [FJS97] discusses an approach to reconstruct base data from summary data and certain constraints, and does not consider the problem of data lineage at the tuple level.

Cui and Buneman in [Cui01], [BKT01] discuss the problem of ambiguity of lineage data. This problem is known as *derivation inequivalence* and arises when equivalent queries have different data lineages for identical tracing data. Cui and Buneman discuss this problem in two scenarios: (a) when aggregation functions are used and (b) when where-provenance is traced. In Section 5.7 of this chapter, we investigate when ambiguity of lineage data may happen in our context and we describe how our DLT approach for tracing why-provenance can also be used for tracing where-provenance, so as to reduce the chance of derivation inequivalence occurring.

5.2 Simple IQL

Our data lineage tracing algorithms assume a subset of IQL^c, *simple IQL* (SIQL), as the query language in transformation pathways. More complex IQL^c queries can be encoded as a series of transformations with SIQL queries on intermediate schema constructs. Although illustrated within this particular query language syntax, our DLT algorithms could also be applied to schema transformation pathways involving queries expressed in other query languages supporting operations on set and bag collections.

5.2.1 The SIQL Syntax

SIQL queries have the following syntax where each collection-valued expression, $\mathtt{D}$, $\mathtt{D}_1 \ldots, \mathtt{D}_n$ below must be a base collection or a variable defined by another SIQL query, and each $\mathtt{cv}_1, ..., \mathtt{cv}_n$ is either a constant (*i.e.* string or number) or a variable defined by another SIQL query:

1. $[\mathtt{cv}_1, \mathtt{cv}_2, ..., \mathtt{cv}_n]$
2. `group D`
3. `sort D`
4. `distinct D`
5. `f D`
6. `gc f D`
7. $\mathtt{D}_1 \mathbin{++} \mathtt{D}_2 \mathbin{++} \ldots \mathbin{++} \mathtt{D}_n$
8. $\mathtt{D}_1 \mathbin{--} \mathtt{D}_2$
9. $[\overline{\mathtt{x}}|\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}_1; ...; \mathtt{C}_k]$
10. $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;\ \mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}}]$
11. $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;\ \mathtt{not}\ (\mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}})]$
12. $\mathtt{map}\ (\mathtt{lambda}\ \mathtt{p}_1.\mathtt{p}_2)\ \mathtt{D}$

SIQL comprehensions are of three forms: $[\overline{\mathtt{x}}|\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}_1; ...; \mathtt{C}_k]$, $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;\ \mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}}]$, and $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;\ \mathtt{not}\ (\mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}})]$. Here, each $\overline{\mathtt{x}_1}$, ..., $\overline{\mathtt{x}_n}$ is either a single variable or a pattern consisting only of variables. $\overline{\mathtt{x}}$ is either a single variable or value, or a pattern of variables or values, and must include all the variables appearing in $\overline{\mathtt{x}_1}$, ..., $\overline{\mathtt{x}_n}$. Each $\mathtt{C}_1$, ..., $\mathtt{C}_k$ is a condition not referring to any base collection. Each variable appearing in $\overline{\mathtt{x}}$ and $\mathtt{C}_1$, ..., $\mathtt{C}_k$ must also appear in some $\overline{\mathtt{x}_i}$, and the variables in $\overline{\mathtt{y}}$ must appear in $\overline{\mathtt{x}}$.

For example, we can use following transformation steps to express a general SPJ operation, $\pi_A(\sigma_C(\mathtt{D}_1 \bowtie ... \bowtie \mathtt{D}_n))$, in SIQL, where $\overline{\mathtt{x}}$ contains all variables appearing in $\overline{\mathtt{x}_1} \ldots \overline{\mathtt{x}_n}$:

$$\mathtt{v1} = [\overline{\mathtt{x}}|\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}]$$
$$\mathtt{v}\ \ = \mathtt{map}\ (\mathtt{lambda}\ \overline{\mathtt{x}}.\mathtt{A})\ \mathtt{v1}$$

Similarly, an aggregate expression $\alpha_{\mathtt{A2},\mathtt{f}(\mathtt{A3})}(\mathtt{D})$ over a collection `D(A1,A2,A3)` is expressed in SIQL as:

```
v1 = map (lambda {x1,x2,x3}.{x2,x3}) D
v  = gc f v1
```

5.2.2 Decomposing IQL^c into SIQL Queries

The syntax of IQL^c and SIQL queries are similar except that the collection-valued expressions in IQL^c queries may be sub-IQL^c queries, while the collection-valued expressions in SIQL queries must be a base collection or a variable defined by another SIQL query. In order to trace data lineage along transformation pathways including general IQL^c queries, we decompose each IQL^c query into a sequence of SIQL queries by means of a depth-first traversal of the IQL^c query tree. This section presents the rules of decomposing IQL^c queries. The algorithms implementing these rules will be discussed in Appendix C. Here, we firstly give an example to show how a general IQL^c query can be decomposed.

Suppose that a view `v` is defined by an IQL^c query `D1` ++ [`{x,z}`|`{x,y}` ← (`D2` −− `D3`); `z` ← [`p|p` ← `D4`, `member D5 p`]; `z < y`]. After decomposing the query, the view definition is expressed by a sequence of SIQL queries as follows:

```
v1  =  D2 −− D3
v2  =  [p|p ← D4, member D5 p]
v3  =  [{x,y,z}|{x,y} ← v1;z ← v2;z < y]
v4  =  map (lambda {x,y,z}.{x,z}) v3
v   =  D1 ++ v4
```

For decomposing IQL^c queries into SIQL queries, we classify IQL^c queries into following four types: 1-*argument queries*, 2-*argument queries*, *n-argument queries*, and *list queries*. The decomposition rules for each type of IQL^c query are as follows:

Decomposition rules for 1-argument queries If an IQL^c query is a 1-argument query, *i.e.*, `group E`, `sort E`, `distinct E`, `aggFun E`, `gc aggFun E` and `map (lambda p1.p2) E`, we decompose the query using following steps:

(1) If `E` is a base collection or a variable, then the query is already a SIQL query

and not required to be decomposed;

(2) If `E` is a sub-query[1], then a new variable is created to replace `E`, and a new transformation step is created to express that the new variable is defined by the replaced sub-query. For example, if `E` is a sub-query, view `v = group E` is decomposed as:

```
v1  =  E
v   =  group v1
```

Decomposition rules for 2-argument queries If an IQL^c query is a 2-argument query, *i.e.* `E1 −− E2`, similar decomposition steps as above are used to decompose the query. However, in this case, we need consider separately the two collection-valued expressions, `E1` and `E2`. For example, if `E1` and `E2` are sub-queries, query `v = E1 −− E2` is decomposed as:

```
v1  =  E1
v2  =  E2
v   =  v1 −− v2
```

Decomposition rules for n-argument queries If an IQL^c query is a n-argument query, *i.e.* an ++ expression or a comprehension, the decomposition rules are as follows:

(1) If the query is an expression of the form `E1 ++ E2 ++ ... ++ En`, the decomposition steps are similar to decomposing 1- and 2-argument queries above, except that each collection-valued expression $\texttt{Ei}(1 \leq \texttt{i} \leq \texttt{n})$ has to be considered separately.

(2) If the query is a comprehension of the form $[\texttt{p}|\texttt{Q1};\ldots;\texttt{Qn}]$, we can refine

[1]Without loss of generality, we assume that a sub-query of an IQL^c query is a SIQL query, since we can recursively decompose the sub-query if it is a general IQL^c query.

this syntax as $[\mathtt{p}|\mathtt{G1};\ldots;\mathtt{Gr};\mathtt{M1};...;\mathtt{Ms};\mathtt{C1};...;\mathtt{Ct}]$, in which $\mathtt{G1}\ldots\mathtt{Gr}$ are generators, $\mathtt{M1}\ldots\mathtt{Ms}$ are filters involving the member function (which we term member *filters*) and $\mathtt{C1}\ldots\mathtt{Ct}$ are filters involving variables, constants and comparison operators (which we term *simple filters*). We recall that each generator `Gi` has syntax $\overline{\mathtt{x}_i} \leftarrow \mathtt{E}_i$ $(1 \leq i \leq \mathtt{r})$ where $\overline{\mathtt{x}_i}$ is a pattern and $\mathtt{E}_i$ is a collection-valued expression.

We first check if the head expression `p` is a pattern containing all the variables appearing in the generator patterns $\overline{\mathtt{x}_i}$ $(1 \leq \mathtt{i} \leq \mathtt{r})$ of the comprehension (we term such comprehensions *select-join comprehensions*). If not, the following intermediate view definitions can be used to transform the comprehension into this form, where $\overline{\mathtt{x}}$ is a pattern containing all the variables appearing in all the generator patterns:

$$\begin{aligned} \mathtt{v1} &= [\overline{\mathtt{x}}|\mathtt{G1};\ldots;\mathtt{Gr};\mathtt{M1};...;\mathtt{Ms};\mathtt{C1};...;\mathtt{Ct}] \\ \mathtt{v} &= \mathtt{map}\ (\mathtt{lambda}\ \overline{\mathtt{x}}.\mathtt{p})\ \mathtt{v1} \end{aligned}$$

In order to decompose the comprehension defining `v1`, we consider each generator and filter.

A generator has the syntax $\overline{\mathtt{x}_i} \leftarrow \mathtt{E}_i$ where $\mathtt{E}_i$ is a collection-valued expression which may be a sub-query. If $\mathtt{E}_i$ is a base collection or a variable, the generator satisfies the SIQL syntax. If $\mathtt{E}_i$ is a sub-query, we redefine the generator in the same way as for decomposing an 1-argument query.

Member filters contain a collection-valued expression `E` which may be a sub-query. Such filters can be redefined in the same way as for decomposing an 1-argument query if the collection-valued expression `E` is a sub-query rather than a base collection or variable.

Furthermore, in the SIQL syntax, there can only be one generator in a comprehension if it contains a member filter, *i.e.* $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{E1};\ \mathtt{member\ E2}\ \overline{\mathtt{y}}]$ and

$[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{E1};\ \mathtt{not\ (member\ E2}\ \overline{\mathtt{y}}\mathtt{)}]$. If a general comprehension contains multiple generators and member filters, we use following decomposition steps to decompose a view v defined by a comprehension $[\overline{\mathtt{x}}|\mathtt{G1};\ldots;\mathtt{Gr};\mathtt{M1};...;\mathtt{Ms};\mathtt{C1};...;\mathtt{Ct}]$ into a sequence of SIQL comprehensions:

$$\begin{aligned}
\mathtt{v}_1 &= [\overline{\mathtt{x}}|\mathtt{G1};\ldots;\mathtt{Gr};\mathtt{C1};...;\mathtt{Ct}]\\
\mathtt{v}_2 &= [\mathtt{p}|\mathtt{p} \leftarrow \mathtt{v}_1;\mathtt{M1}]\\
\mathtt{v}_3 &= [\mathtt{p}|\mathtt{p} \leftarrow \mathtt{v}_2;\mathtt{M2}]\\
&\ldots\\
\mathtt{v} &= [\mathtt{p}|\mathtt{p} \leftarrow \mathtt{v}_s;\mathtt{Ms}]
\end{aligned}$$

To illustrate the whole decomposition process for a comprehension, suppose that the view `v` is defined by the comprehension [`{x,z}| {x,y}` ← `D1;z` ← `(D2 ++ D3);member (D4 -- D5) z;not (member D6 {y,z}); x>z`]. This view definition is decomposed into following SIQL queries:

```
v1  =  D2 ++ D3
v2  =  [{x,y,z}|{x,y} ← D1; z ← v1; x>z]
v3  =  D4 -- D5
v4  =  [{x,y,z}|{x,y,z} ← v2; member v3 z]
v5  =  [{x,y,z}|{x,y,z} ← v4; not (member D6 {y,z})]
v   =  map (lambda {x,y,z}.{x,z}) v5
```

Decomposition rules for list expressions In IQL^c, there may be list expressions which contain IQL^c sub-queries. If the query is a list expression, $[\mathsf{e}_1, \mathsf{e}_2, ..., \mathsf{e}_n]$, this may be a list containing only constants, such as `[1,2,3,4]`, or a list containing sub-queries as its items, such as `[1,2,max [2,3,4],sum [3,4,5]]`. In the former case, there is no need to decompose it. In the latter case, without loss of generality, the general form of such a query is

$$[\mathsf{c}_1, ..., \mathsf{c}_r, \mathsf{e}_1, ..., \mathsf{e}_s]$$

in which $\mathtt{c}_1, ..., \mathtt{c}_r$ are constants and $\mathtt{e}_1, ..., \mathtt{e}_s$ are sub-queries. Note that, we do not consider the order of items in a list in IQL^c, *i.e.* lists here have the semantics of bags. The above query can be expressed by the following ++ expression:

$$[\mathtt{c}_1, ..., \mathtt{c}_r] \mathbin{++} [\mathtt{e}_1] \mathbin{++} \ldots \mathbin{++} [\mathtt{e}_s]$$

and each $\mathtt{e}_i$ $(1 \leq i \leq s)$ can then be further decomposed. For example, suppose that the view `v` is defined by the query `[1,2,max [2,3,4],sum [3,4,5]]`. Then `v` can be expressed by following SIQL queries:

```
v1  =  max [2,3,4]
v2  =  sum [3,4,5]
v3  =  [1,2]
v4  =  [v1]
v5  =  [v2]
v   =  v3 ++ v4 ++ v5
```

Suppose a view `v` is defined by a list expression. If the list expression can be transformed as above into a ++ expression, the problem of tracing `v`'s lineage or of incrementally maintaining `v` is subsumed by considering the ++ expression. If the list expression cannot be transformed into a ++ expression, then the list is a list of constants; the lineage data will be the tracing data itself, and the view cannot be updated. Thus, in the rest of this thesis, we do not consider the case of list expressions for data lineage tracing or for incremental view maintenance.

5.2.3 An Example of Schema Transformations

Consider two relational schemas `SS` and `GS`. `SS` is a source schema containing two relations mathematician(<u>emp_id</u>, `salary`) and compScientist(<u>emp_id</u>, `salary`). `GS` is the target schema containing two relations person(<u>emp_id</u>, `salary`, `dept`) and department($\underline{\mathsf{deptName}}$, `avgDeptSalary`).

By the definition of our simple relational model, `SS` has a set of Rel constructs Rel_1 and a set of Att constructs Att_1, while `GS` has a set of Rel constructs Rel_2 and a set of Att constructs Att_2, where:

$$\begin{aligned}
Rel_1 &= \{\langle\langle \mathsf{mathematician}\rangle\rangle, \langle\langle \mathsf{compScientist}\rangle\rangle\} \\
Att_1 &= \{\langle\langle \mathsf{mathematician}, \mathsf{emp\_id}\rangle\rangle, \langle\langle \mathsf{mathematician}, \mathsf{salary}\rangle\rangle \\
&\quad \langle\langle \mathsf{compScientist}, \mathsf{emp\_id}\rangle\rangle, \langle\langle \mathsf{compScientist}, \mathsf{salary}\rangle\rangle\} \\
Rel_2 &= \{\langle\langle \mathsf{person}\rangle\rangle, \langle\langle \mathsf{department}\rangle\rangle\} \\
Att_2 &= \{\langle\langle \mathsf{person}, \mathsf{emp\_id}\rangle\rangle, \langle\langle \mathsf{person}, \mathsf{salary}\rangle\rangle, \langle\langle \mathsf{person}, \mathsf{dept}\rangle\rangle \\
&\quad \langle\langle \mathsf{department}, \mathsf{deptName}\rangle\rangle, \langle\langle \mathsf{department}, \mathsf{avgDeptSalary}\rangle\rangle\}
\end{aligned}$$

Schema `SS` can be transformed to `GS` by the sequence of primitive schema transformations given below. The first seven transformation steps create the constructs of `GS` which do not exist in `SS`. The query in each step gives the extension of the new schema construct in terms of the extents of the existing schema constructs. The last six steps then delete the redundant constructs of `SS`. The query in each of these steps shows how the extension of each deleted construct can be reconstructed from the remaining schema constructs:

(1) addRel $(\langle\langle \mathsf{person}\rangle\rangle, \langle\langle \mathsf{mathematician}\rangle\rangle \mathbin{++} \langle\langle \mathsf{compScientist}\rangle\rangle)$;

(2) addAtt $(\langle\langle \mathsf{person}, \mathsf{emp\_id}\rangle\rangle, \langle\langle \mathsf{mathematician}, \mathsf{emp\_id}\rangle\rangle \mathbin{++} \langle\langle \mathsf{compScientist}, \mathsf{emp\_id}\rangle\rangle)$;

(3) addAtt $(\langle\langle \mathsf{person}, \mathsf{salary}\rangle\rangle, \langle\langle \mathsf{mathematician}, \mathsf{salary}\rangle\rangle \mathbin{++} \langle\langle \mathsf{compScientist}, \mathsf{salary}\rangle\rangle)$;

(4) addAtt $(\langle\langle \mathsf{person}, \mathsf{dept}\rangle\rangle, [\{x, \texttt{'Maths'}\} | x \leftarrow \langle\langle \mathsf{mathematician}\rangle\rangle] \mathbin{++}$
$[\{x, \texttt{'CompSci'}\} | x \leftarrow \langle\langle \mathsf{compScientist}\rangle\rangle])$;

(5) addRel $(\langle\langle \mathsf{department}\rangle\rangle, [\texttt{'Maths'}, \texttt{'CompSci'}])$;

(6) addAtt $(\langle\langle \mathsf{department}, \mathsf{deptName}\rangle\rangle, [\{\texttt{'Maths'}, \texttt{'Maths'}\}, \{\texttt{'CompSci'}, \texttt{'CompSci'}\}])$;

(7) addAtt $(\langle\langle \mathsf{department}, \mathsf{avgDeptSalary}\rangle\rangle,$
$\texttt{gc avg } [\{\texttt{'Maths'}, s\} | \{x, s\} \leftarrow \langle\langle \mathsf{mathematician}, \mathsf{salary}\rangle\rangle] \mathbin{++}$
$\texttt{gc avg } [\{\texttt{'Maths'}, s\} | \{x, s\} \leftarrow \langle\langle \mathsf{mathematician}, \mathsf{salary}\rangle\rangle])$;

(8) delAtt $(\langle\langle\mathsf{mathematician, salary}\rangle\rangle, [\{x, s\}|\{x, s\} \leftarrow \langle\langle\mathsf{person, salary}\rangle\rangle;$
$\{x', d\} \leftarrow \langle\langle\mathsf{person, dept}\rangle\rangle; d = \texttt{'Maths'}; x = x'])$;

(9) delAtt $(\langle\langle\mathsf{mathematician, emp\_id}\rangle\rangle, [\{x, id\}|\{x, id\} \leftarrow \langle\langle\mathsf{person, emp\_id}\rangle\rangle;$
$\{x', d\} \leftarrow \langle\langle\mathsf{person, dept}\rangle\rangle; d = \texttt{'Maths'}; x = x'])$;

(10) delRel $(\langle\langle\mathsf{mathematician}\rangle\rangle, [x|\{x, d\} \leftarrow \langle\langle\mathsf{person, dept}\rangle\rangle; d = \texttt{'Maths'}])$;

(11) delAtt $(\langle\langle\mathsf{compScientist, salary}\rangle\rangle, [\{x, s\}|\{x, s\} \leftarrow \langle\langle\mathsf{person, salary}\rangle\rangle;$
$\{x', d\} \leftarrow \langle\langle\mathsf{person, dept}\rangle\rangle; d = \texttt{'CompSci'}; x = x'])$;

(12) delAtt $(\langle\langle\mathsf{compScientist, emp\_id}\rangle\rangle, [\{x, id\}|\{x, id\} \leftarrow \langle\langle\mathsf{person, emp\_id}\rangle\rangle;$
$\{x', d\} \leftarrow \langle\langle\mathsf{person, dept}\rangle\rangle; d = \texttt{'CompSci'}; x = x'])$;

(13) delRel $(\langle\langle\mathsf{compScientist}\rangle\rangle, [x|\{x, d\} \leftarrow \langle\langle\mathsf{person, dept}\rangle\rangle; d = \texttt{'CompSci'}])$;

IQL^c queries are automatically broken down by our data lineage tracing software into a sequence of add or delete transformations with SIQL queries within them. The decomposition procedure undertakes a depth-first search of the query tree and generates the sequence of transformations from the bottom up. For example, the following decompositions would be equivalent to steps (4) and (7) above, with (4.1) $\sim$ (4.5) replacing step (4) and (7.1) $\sim$ (7.9) replacing step[2]:

(4.1) addAtt $(\texttt{\$Query\_4\_1}, [\{x, \texttt{'Maths'}\}|x \leftarrow \langle\langle\mathsf{mathematician}\rangle\rangle])$;

(4.2) addAtt $(\texttt{\$Query\_4\_2}, [\{x, \texttt{'CompSci'}\}|x \leftarrow \langle\langle\mathsf{compScientist}\rangle\rangle])$;

(4.3) addAtt $(\langle\langle\mathsf{person, dept}\rangle\rangle, \texttt{\$Query\_4\_1} +\!\!+ \texttt{\$Query\_4\_2})$;

(4.4) delAtt $(\texttt{\$Query\_4\_2}, [\{x, \texttt{'CompSci'}\}|x \leftarrow \langle\langle\mathsf{compScientist}\rangle\rangle])$;

(4.5) delAtt $(\texttt{\$Query\_4\_1}, [\{x, \texttt{'Maths'}\}|x \leftarrow \langle\langle\mathsf{mathematician}\rangle\rangle])$;

[2]Note that, the intermediate construct names $Query_i_j are automatically generated by our IQL^c decomposition algorithms

(7.1) addRel ($\texttt{\$Query\_7\_1}$,

$\texttt{map (lambda } \{x,s\}.\{'\texttt{Maths}', s\})\ \langle\langle \textsf{mathematician}, \textsf{salary}\rangle\rangle$);

(7.2) addRel ($\texttt{\$Query\_7\_2}$, $\texttt{gc avg \$Query\_7\_1}$);

(7.3) addRel ($\texttt{\$Query\_7\_3}$,

$\texttt{map (lambda } \{x,s\}.\{'\texttt{CompSci}', s\})\ \langle\langle \textsf{compScientist}, \textsf{salary}\rangle\rangle$);

(7.4) addRel ($\texttt{\$Query\_7\_4}$, $\texttt{gc avg \$Query\_7\_3}$);

(7.5) addAtt ($\langle\langle \textsf{department}, \textsf{avgDeptSalary}\rangle\rangle$, $\texttt{\$Query\_7\_2} \mathbin{++} \texttt{\$Query\_7\_4}$);

(7.6) delRel ($\texttt{\$Query\_7\_4}$, $\texttt{gc avg \$Query\_7\_3}$);

(7.7) delRel ($\texttt{\$Query\_7\_3}$,

$\texttt{map (lambda } \{x,s\}.\{'\texttt{CompSci}', s\})\ \langle\langle \textsf{compScientist}, \textsf{salary}\rangle\rangle$);

(7.8) delRel ($\texttt{\$Query\_7\_2}$, $\texttt{gc avg \$Query\_7\_1}$);

(7.9) delRel ($\texttt{\$Query\_7\_1}$,

$\texttt{map (lambda } \{x,s\}.\{'\texttt{Maths}', s\})\ \langle\langle \textsf{mathematician}, \textsf{salary}\rangle\rangle$);

5.3 Data Lineage Definitions

We consider both *affect-provenance* and *origin-provenance* in our treatment of the data lineage tracing problem. What we regard as affect-provenance includes all of the source data that had some influence on the tracing data. Origin-provenance is simpler because here we are only interested in the specific data in the sources from which the tracing data is extracted. In particular, we use the notions of *maximal witness* and *minimal witness* from [BKT01] to define the notions of *affect-pool* and *origin-pool*, respectively, in Definitions 1 and 2 below, and we use a condition from [CWW00] to guarantee that there are no redundant elements in the computed lineage data.

In both these definitions, $\texttt{v} = \texttt{q}(\mathbb{D})$ is a view over a set of bags $\mathbb{D}$ defined by the query $\texttt{q}$ and $t \in \texttt{v}$ is a tracing tuple. Condition (a) states that the result of applying query $\mathtt{q}$ to the lineage data must be the bag consisting of all copies of t in the view $\mathtt{v}$. Condition (b) is used to enforce the maximizing and minimizing properties, respectively. Thus, the affect-pool includes all elements in the data sources which could generate t by applying $\mathtt{q}$ to them; conversely, if any element and all of its copies in the origin-pool was deleted, then t or all of t's copies in $\mathtt{v}$ could not be generated by applying the query $\mathtt{q}$ to the lineage data. Condition (c) guarantees that there are no redundant elements in the computed lineage data. Condition (d) in Definition 2 ensures that if the origin-pool of the tracing tuple t in the source bag $\mathtt{D}_i$ is $\mathtt{T}_i^{op}$, then for any tuple in $\mathtt{D}_i$, either all of the copies of the tuple are in $\mathtt{T}_i^{op}$ or none of them are in $\mathtt{T}_i^{op}$.

Note that, both the definitions apply to tracing data lineage for a single SIQL query. For a view created by a sequence of SIQL queries, we have additional data lineage definitions which we give in Section 5.5.1 below.

Definition 1 **(Affect-pool for a SIQL query)** Let $\mathtt{q}$ be any SIQL query over bags $\mathtt{D}_1, \ldots, \mathtt{D}_m$, and let $\mathtt{v} = \mathtt{q}(\mathtt{D}_1, \ldots, \mathtt{D}_m)$ be the bag that results from applying $\mathtt{q}$ to $\mathtt{D}_1, \ldots, \mathtt{D}_m$. Given a tracing tuple $t \in \mathtt{v}$, we define t's *affect-pool* in $\mathtt{D}_1, \ldots, \mathtt{D}_m$ *according to* $\mathtt{q}$, $\mathtt{q}^{AP}_{\langle \mathtt{D}_1,\ldots,\mathtt{D}_m\rangle}(t)$, to be the sequence of bags $\langle \mathtt{T}_1^{ap}, \ldots, \mathtt{T}_m^{ap}\rangle$, where $\mathtt{T}_1^{ap}, \ldots, \mathtt{T}_m^{ap}$ are *maximal* sub-bags of $\mathtt{D}_1, \ldots, \mathtt{D}_m$ such that:

(a) $\mathtt{q}(\mathtt{T}_1^{ap}, \ldots, \mathtt{T}_m^{ap}) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{v};\ \mathtt{x} = t]$

(b) $\forall \mathtt{T}'_1 \subseteq \mathtt{D}_1, \ldots, \mathtt{T}'_m \subseteq \mathtt{D}_m$: $\mathtt{q}(\mathtt{T}'_1, \ldots, \mathtt{T}'_m) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{v};\ \mathtt{x} = \mathtt{t}]$

$\Rightarrow \mathtt{T}'_1 \subseteq \mathtt{T}_1^{ap}, \ldots, \mathtt{T}'_m \subseteq \mathtt{T}_m^{ap}$

(c) $\forall \mathtt{T}_i^{ap}$: $\forall t^* \in \mathtt{T}_i^{ap}$: $\mathtt{q}(\mathtt{T}_1^{ap}, \ldots, [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{T}_i^{ap}; \mathtt{x} = t^*], \ldots, \mathtt{T}_m^{ap}) \neq \emptyset$

We say that $\mathtt{q}^{AP}_{\mathtt{D}_i}(t) = \mathtt{T}_i^{ap}$ is *t's affect-pool in* $\mathtt{D}_i$.

Definition 2 (Origin-pool for a SIQL query) Let $\mathtt{q}$, $\mathtt{D}_1$, …, $\mathtt{D}_m$, t, $\mathtt{v}$ and $\mathtt{q}$ be as above. We define t's *origin-pool in* $\mathtt{D}_1$, …, $\mathtt{D}_m$ *according to* $\mathtt{q}$, $\mathtt{q}^{OP}_{\langle \mathtt{D}_1,\ldots,\mathtt{D}_m\rangle}(t)$, to be the sequence of bags $\langle \mathtt{T}^{op}_1, \ldots, \mathtt{T}^{op}_m\rangle$, where $\mathtt{T}^{op}_1$, …, $\mathtt{T}^{op}_m$ are *minimal* sub-bags of $\mathtt{D}_1$, …, $\mathtt{D}_m$ such that:

(a) $\mathtt{q}(\mathtt{T}^{op}_1, \ldots, \mathtt{T}^{op}_m) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{v};\ \mathtt{x} = t]$

(b) $\forall \mathtt{T}^{op}_i$: $\forall t^* \in \mathtt{T}^{op}_i$: $\mathtt{q}(\mathtt{T}^{op}_1, \ldots, [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{T}^{op}_i;\ \mathtt{x} \neq t^*], \ldots, \mathtt{T}^{op}_m) \neq [\ \mathtt{x}|\mathtt{x} \leftarrow \mathtt{v};\ \mathtt{x} = t]$

(c) $\forall \mathtt{T}^{op}_i$: $\forall t^* \in \mathtt{T}^{op}_i$: $\mathtt{q}(\mathtt{T}^{op}_1, \ldots, [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{T}^{op}_i; \mathtt{x} = t^*], \ldots, \mathtt{T}^{op}_m) \neq \emptyset$

(d) $\forall \mathtt{T}^{op}_i$: $\forall t^* \in \mathtt{T}^{op}_i$: $t^* \notin (\mathtt{D}_i -\!- \mathtt{T}^{op}_i)$

We say that $\mathtt{q}^{OP}_{\mathtt{D}_i}(t) = \mathtt{T}^{op}_i$ is *t's origin-pool in* $\mathtt{D}_i$.

Proposition 1. Suppose that the affect-pool and origin-pool of a tracing tuple t is the sequence of bags $\langle \mathtt{T}^{ap}_1, \ldots, \mathtt{T}^{ap}_m\rangle$ and the sequence of bags $\langle \mathtt{T}^{op}_1, \ldots, \mathtt{T}^{op}_m\rangle$, respectively, then each bag $\mathtt{T}^{op}_i$ is a sub-bag of $\mathtt{T}^{ap}_i$.

The condition (b) in Definition 1 ensures that, for any sequence of bags $\langle \mathtt{T}'_1$, …, $\mathtt{T}'_m\rangle$, if $\mathtt{q}(\mathtt{T}^{op}_1, \ldots, \mathtt{T}^{op}_m) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{v};\ \mathtt{x} = t]$, then each bag $\mathtt{T}'_i$ is a sub-bag of $\mathtt{T}^{ap}_i$. Thus, from condition (a) in Definition 2, each bag $\mathtt{T}^{op}_i$ is a sub-bag of $\mathtt{T}^{ap}_i$.

5.4 Data Lineage Tracing Formulae

Following on from the above definitions of data lineage and the definition of SIQL queries in Section 5.2, we now specify the affect-pool and origin-pool for SIQL queries. As in [CWW00], we use *derivation tracing queries* to evaluate the lineage of a tuple t or a set of tuples T with respect to a set of bags $\mathbb{D}$. That is, we apply a query to $\mathbb{D}$ and the result is the derivation of t (or T) in $\mathbb{D}$. We call such a query the *tracing query for t (or T) on* $\mathbb{D}$, denoted as $\mathtt{TQ}_{\mathbb{D}}(t)$ (or $\mathtt{TQ}_{\mathbb{D}}(T)$).

Theorem 1 (*Affect-pool* and *Origin-pool* for a tuple with SIQL queries). Let $\mathtt{v} = \mathtt{q}(\mathbb{D})$ be the bag that results from applying a SIQL query $\mathtt{q}$ to a sequence of bags $\mathbb{D}$. Then, for any tuple $t \in \mathtt{v}$, the tracing queries $\mathtt{TQ}^{AP}_{\mathbb{D}}(t)$ below give the affect-pool of t in $\mathbb{D}$, and the tracing queries $\mathtt{TQ}^{OP}_{\mathbb{D}}(t)$ give the origin-pool of t in $\mathbb{D}$:

$$\begin{array}{lrcl}
\text{t1:} & \mathtt{q} & = & \mathtt{D}_1 \mathbin{++} \ldots \mathbin{++} \mathtt{D}_n \qquad (\mathbb{D} = \langle \mathtt{D}_1, \ldots, \mathtt{D}_n \rangle) \\
 & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) \;= \langle [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}_1; \mathtt{x} = t], \ldots, [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}_n; \mathtt{x} = t] \rangle \\
\text{t2:} & \mathtt{q} & = & \mathtt{D}_1 \mathbin{--} \mathtt{D}_2 \qquad (\mathbb{D} = \langle \mathtt{D}_1, \mathtt{D}_2 \rangle) \\
 & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \langle [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}_1; \mathtt{x} = t], \mathtt{D}_2 \rangle \\
 & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) & = & \langle [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}_1; \mathtt{x} = t], [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}_2; \mathtt{x} = t] \rangle \\
\text{t3:} & \mathtt{q} & = & \mathtt{group\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle) \\
 & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) \;= [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; (\mathtt{first\ x}) = (\mathtt{first}\ t)] \\
\text{t4:} & \mathtt{q} & = & \mathtt{sort\ D/\ distinct\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle) \\
 & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) \;= [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{x} = t] \\
\text{t5:} & \mathtt{q} & = & \mathtt{max\ D\ /\ min\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle) \\
 & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{D} \\
 & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) & = & [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{x} = t] \\
\text{t6:} & \mathtt{q} & = & \mathtt{sum\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle) \\
 & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{D} \\
 & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) & = & [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{x} \neq 0] \\
\text{t7:} & \mathtt{q} & = & \mathtt{count\ D\ /\ avg\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle) \\
 & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) \;= \mathtt{D} \\
\text{t8:} & \mathtt{q} & = & \mathtt{gc\ max\ D\ /\ gc\ min\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle) \\
 & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; (\mathtt{first\ x}) = (\mathtt{first}\ t)] \\
 & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) & = & [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{x} = t] \\
\text{t9:} & \mathtt{q} & = & \mathtt{gc\ sum\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle) \\
 & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; (\mathtt{first\ x}) = (\mathtt{first}\ t)] \\
 & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) & = & [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; (\mathtt{first\ x}) = (\mathtt{first}\ t); (\mathtt{second\ x}) \neq 0]
\end{array}$$

$$
\begin{array}{rrcl}
\text{t10:} & \mathtt{q} & = & \mathtt{gc\ count\ D\ /\ gc\ avg\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle) \\
& \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) \ = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; (\mathtt{first\ x}) = (\mathtt{first}\ t)] \\
\text{t11:} & \mathtt{q} & = & [\overline{\mathtt{x}}|\overline{\mathtt{x_1}} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}_1; \ldots; \mathtt{C}_k] \qquad (\mathbb{D} = \langle \mathtt{D}_1, \ldots, \mathtt{D}_n \rangle) \\
& \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) \ = \langle [\mathtt{x}_1|\mathtt{x}_1 \leftarrow \mathtt{D}_1; \mathtt{x}_1 = ((\mathtt{lambda}\ \overline{\mathtt{x}}.\overline{\mathtt{x_1}})\ t)], \ldots, \\
& & & \qquad [\mathtt{x}_n|\mathtt{x}_n \leftarrow \mathtt{D}_n; \mathtt{x}_n = ((\mathtt{lambda}\ \overline{\mathtt{x}}.\overline{\mathtt{x}_n})\ t)] \rangle \\
\text{t12:} & \mathtt{q} & = & [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1; \mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}}] \qquad (\mathbb{D} = \langle \mathtt{D}_1, \mathtt{D}_2 \rangle) \\
& \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) \ = \langle [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1; \overline{\mathtt{x}} = t], [\mathtt{y}|\mathtt{y} \leftarrow \mathtt{D}_2; \mathtt{y} = ((\mathtt{lambda}\ \overline{\mathtt{x}}.\overline{\mathtt{y}})\ t)] \rangle \\
\text{t13:} & \mathtt{q} & = & [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1; \mathtt{not}\ (\mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}})] \qquad (\mathbb{D} = \langle \mathtt{D}_1, \mathtt{D}_2 \rangle) \\
& \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \langle [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1; \overline{\mathtt{x}} = t], \mathtt{D}_2 \rangle \\
& \mathtt{TQ}^{OP}_{\mathbb{D}}(t) & = & \langle [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1; \overline{\mathtt{x}} = t], \emptyset \rangle \\
\text{t14:} & \mathtt{q} & = & \mathtt{map}\ (\mathtt{lambda}\ \mathtt{p}_1.\mathtt{p}_2)\ \mathtt{D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle) \\
& \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) = [\mathtt{p}_1|\mathtt{p}_1 \leftarrow \mathtt{D}; \mathtt{p}_2 = t]
\end{array}
$$

We note that general IQL^c queries are allowed in the tracing queries. Appendix *A* gives the proof that the results of queries $\mathtt{TQ}^{AP}_{\mathbb{D}}(t)$ and $\mathtt{TQ}^{OP}_{\mathbb{D}}(t)$ in Theorem 1 satisfy Definition 1 and 2 respectively.

Theorem 2 (*Affect-pool* and *Origin-pool* for a set of tuples with SIQL queries). Let $\mathtt{v} = \mathtt{q}(\mathbb{D})$ be the bag that results from applying a SIQL query $\mathtt{q}$ to a sequence of bags $\mathbb{D}$. Then, for a set of tuples $T \subseteq \mathtt{v}$ ($T \neq \emptyset$), the tracing queries $\mathtt{TQ}^{AP}_{\mathbb{D}}(T)$ below give the affect-pool of T in $\mathbb{D}$, and the tracing queries $\mathtt{TQ}^{OP}_{\mathbb{D}}(T)$ give the origin-pool of T in $\mathbb{D}$:

$$
\begin{array}{rrcl}
\text{T1:} & \mathtt{q} & = & \mathtt{D}_1 +\!\!+ \ldots +\!\!+ \mathtt{D}_n \qquad (\mathbb{D} = \langle \mathtt{D}_1, \ldots, \mathtt{D}_n \rangle) \\
& \mathtt{TQ}^{AP}_{\mathbb{D}}(T) & = & \mathtt{TQ}^{OP}_{\mathbb{D}}(T) \ = \langle [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}_1; \mathtt{member}\ T\ \mathtt{x}], \ldots, \\
& & & \qquad\qquad [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}_n; \mathtt{member}\ T\ \mathtt{x}] \rangle \\
\text{T2:} & \mathtt{q} & = & \mathtt{D}_1 -\!- \mathtt{D}_2 \qquad (\mathbb{D} = \langle \mathtt{D}_1, \mathtt{D}_2 \rangle) \\
& \mathtt{TQ}^{AP}_{\mathbb{D}}(T) & = & \langle [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}_1; \mathtt{member}\ T\ \mathtt{x}], \mathtt{D}_2 \rangle \\
& \mathtt{TQ}^{OP}_{\mathbb{D}}(T) & = & \langle [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}_1; \mathtt{member}\ T\ \mathtt{x}], [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}_2; \mathtt{member}\ T\ \mathtt{x}] \rangle
\end{array}
$$

T3: $\mathtt{q} = \mathtt{group\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle)$

$$\mathtt{TQ}^{AP}_{\mathbb{D}}(T) = \mathtt{TQ}^{OP}_{\mathbb{D}}(T) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{member}\ [(\mathtt{first\ y})|\mathtt{y} \leftarrow T]\ (\mathtt{first\ x})]$$

T4: $\mathtt{q} = \mathtt{sort\ D/\ distinct\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle)$

$$\mathtt{TQ}^{AP}_{\mathbb{D}}(T) = \mathtt{TQ}^{OP}_{\mathbb{D}}(T) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{member}\ T\ \mathtt{x}]$$

T5: $\mathtt{q} = \mathtt{f\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle)$

N/A /* The tracing data cannot be a set of tuples */

T6: $\mathtt{q} = \mathtt{gc\ max\ D\ /\ gc\ min\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle)$

$$\mathtt{TQ}^{AP}_{\mathbb{D}}(T) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{member}\ [(\mathtt{first\ y})|\mathtt{y} \leftarrow T]\ (\mathtt{first\ x})]$$

$$\mathtt{TQ}^{OP}_{\mathbb{D}}(T) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{member}\ T\ \mathtt{x}]$$

T7: $\mathtt{q} = \mathtt{gc\ sum\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle)$

$$\mathtt{TQ}^{AP}_{\mathbb{D}}(T) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{member}\ [(\mathtt{first\ y})|\mathtt{y} \leftarrow T]\ (\mathtt{first\ x})]$$

$$\mathtt{TQ}^{OP}_{\mathbb{D}}(T) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{member}\ [(\mathtt{first\ y})|\mathtt{y} \leftarrow T]\ (\mathtt{first\ x}); (\mathtt{second\ x}) \neq 0]$$

T8: $\mathtt{q} = \mathtt{gc\ count\ D\ /\ gc\ avg\ D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle)$

$$\mathtt{TQ}^{AP}_{\mathbb{D}}(T) = \mathtt{TQ}^{OP}_{\mathbb{D}}(T) = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D}; \mathtt{member}\ [(\mathtt{first\ y})\,|\,\mathtt{y} \leftarrow T]\ (\mathtt{first\ x})]$$

T9: $\mathtt{q} = [\overline{\mathtt{x}}|\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}_1; \ldots; \mathtt{C}_k] \qquad (\mathbb{D} = \langle \mathtt{D}_1, \ldots, \mathtt{D}_n \rangle)$

$$\mathtt{TQ}^{AP}_{\mathbb{D}}(T) = \mathtt{TQ}^{OP}_{\mathbb{D}}(T) = \langle [\mathtt{x}_1|\mathtt{x}_1 \leftarrow \mathtt{D}_1; \mathtt{member}\ (\mathtt{map}\ (\mathtt{lambda}\ \overline{\mathtt{x}}.\overline{\mathtt{x}_1})\ T)\ \mathtt{x}_1], \ldots, [\mathtt{x}_n|\mathtt{x}_n \leftarrow \mathtt{D}_n; \mathtt{member}\ (\mathtt{map}\ (\mathtt{lambda}\ \overline{\mathtt{x}}.\overline{\mathtt{x}_n})\ T)\ \mathtt{x}_n] \rangle$$

T10: $\mathtt{q} = [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1; \mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}}] \qquad (\mathbb{D} = \langle \mathtt{D}_1, \mathtt{D}_2 \rangle)$

$$\mathtt{TQ}^{AP}_{\mathbb{D}}(T) = \mathtt{TQ}^{OP}_{\mathbb{D}}(T) = \langle [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1; \mathtt{member}\ T\ \overline{\mathtt{x}}], [\mathtt{y}|\mathtt{y} \leftarrow \mathtt{D}_2; \mathtt{member}\ (\mathtt{map}\ (\mathtt{lambda}\ \overline{\mathtt{x}}.\overline{\mathtt{y}})\ T)\ \mathtt{y}] \rangle$$

T11: $\mathtt{q} = [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1; \mathtt{not}\ (\mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}})] \qquad (\mathbb{D} = \langle \mathtt{D}_1, \mathtt{D}_2 \rangle)$

$$\mathtt{TQ}^{AP}_{\mathbb{D}}(T) = \langle [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1; \mathtt{member}\ T\ \overline{\mathtt{x}}], \mathtt{D}_2 \rangle$$

$$\mathtt{TQ}^{OP}_{\mathbb{D}}(T) = \langle [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1; \mathtt{member}\ T\ \overline{\mathtt{x}}], \emptyset \rangle$$

T12: $\quad \mathtt{q} \;=\; \mathtt{map}\,(\mathtt{lambda}\ \mathtt{p}_1.\mathtt{p}_2)\ \mathtt{D} \qquad (\mathbb{D} = \langle \mathtt{D} \rangle)$

$$\mathtt{TQ}_{\mathbb{D}}^{AP}(T) = \mathtt{TQ}_{\mathbb{D}}^{OP}(T) = [\mathtt{p}_1|\mathtt{p}_1 \leftarrow \mathtt{D}; \mathtt{member}\ T\ \mathtt{p}_2]$$

The proof of Theorem 2 is similar to Theorem 1. Note that, if the tracing set T is empty, we assume that T's lineage data is empty as well.

5.5 Data Lineage Tracing Algorithm

Section 5.4 presented formulae for obtaining tracing queries from SIQL queries. This section gives an algorithm for tracing the lineage data of data in a materialised view that has been defined by a transformation pathway from a data source schema. For simplicity of exposition, we assume that all of the data source schemas have first been integrated into a single schema S consisting of the union of the constructs of the individual source schemas, with appropriate renaming of schema constructs to avoid duplicate names.

The DLT algorithm described in this section assumes that all intermediate transformation steps are materialised, *i.e.* the constructs created by add transformation steps are materialised. DLT algorithms for more general transformation pathways will be discussed in Chapter 6.

In general, intermediate constructs created during the IQL^c to SIQL decomposition by an add transformation do not remain in the materialised global schema as they are removed by a delete transformation in the transformation steps after the add transformation. In order to materialise these intermediate constructs, we remove the transformations which delete these intermediate constructs so as to leave them in the materialised global schema.

5.5.1 Tracing Data Lineage through Transformation Pathways

Suppose an integrated schema GS has been derived from a source schema S though a transformation pathway $TP = tp_1, \ldots, tp_r$. Regarding each transformation step as a function applied to S, GS can be obtained as $\mathtt{GS} = tp_1 \circ tp_2 \circ \ldots \circ tp_r(\mathtt{S}) = tp_r(\ldots(tp_2(tp_1(\mathtt{S})))\ldots)$. Thus, tracing the lineage of data in GS requires tracing data lineage via a *query-sequence*, defined as follows:

Definition 3 (Affect-pool for a query-sequence) Let $\mathtt{Q} = \mathtt{q}_1, \mathtt{q}_2, \ldots, \mathtt{q}_r$ be a query-sequence over a sequence of bags $\mathbb{D}$, and let $\mathtt{v} = \mathtt{Q}(\mathbb{D}) = \mathtt{q}_1 \circ \mathtt{q}_2 \circ \ldots \circ \mathtt{q}_r(\mathbb{D})$ be the bag that results from applying Q to $\mathbb{D}$. Given a tracing tuple $t \in \mathtt{v}$, we define t's *affect-pool in* $\mathbb{D}$ *according to* Q, $\mathtt{Q}_{\mathbb{D}}^{AP}(t)$, to be $\mathbb{D}^{ap}$, where $\mathbb{D}_i^{ap} = \mathtt{q}_i^{AP}(\mathbb{D}_{i+1}^{ap})$ $(1 \leq i \leq r)$, $\mathbb{D}_{i+1}^{ap} = \{t\}$ and $\mathbb{D}^{ap} = \mathbb{D}_1^{ap}$.

Definition 4 (Origin-pool for query-sequence) Let Q, $\mathbb{D}$, v and t be as above. We define t's *origin-pool in* $\mathbb{D}$ *according to* Q, $\mathtt{Q}_{\mathbb{D}}^{OP}(t)$, to be $\mathbb{D}^{op}$, where $\mathbb{D}_i^{op} = \mathtt{q}_i^{OP}(\mathbb{D}_{i+1}^{op})$ $(1 \leq i \leq r)$, $\mathbb{D}_{i+1}^{op} = \{t\}$ and $\mathbb{D}^{op} = \mathbb{D}_1^{op}$.

Definitions 3 and 4 state that the derivations of data in an integrated schema GS can be derived by examining the transformation pathways from the source schema S to GS in reverse, step by step.

An AutoMed transformation pathway consists of a sequence of primitive transformations which generate the integrated schema from the given source schemas. The schema constructs are generally different for different modelling languages. When considering data lineage tracing, we are only concerned with structural constructs associated with a data extent e.g. Node and Edge constructs in the HDM, Rel and Att constructs in the simple relational data model, and Element, Attribute and NestSet constructs in the simple XML data model. Thus, for data lineage tracing, we ignore primitive schema transformation steps which are adding, deleting or renaming only constraints. Moreover, we treat any primitive transformation which is adding a construct to a schema as a generic `addT` transformation, any primitive transformation which is deleting a construct from a schema as a generic `delT` transformation, and any primitive transformation which is renaming a schema construct as a generic `renameT` transformation. We can summarise the problem of data lineage for each of these transformations as follows[3]:

(a) For an `addT`(`c`, `q`) transformation, the lineage of data in the extent of schema construct `c` is located in the extents of the schema constructs appearing in the query `q`.

(b) For a `renameT`(`c'`, `c`) transformation, the lineage of data in the extent of schema construct `c` is located in the extent of schema construct `c'`.

(c) All `delT`(`c`, `q`) transformations can be ignored since they create no schema constructs.

5.5.2 Algorithms for Tracing Data Lineage

In our algorithms below, we assume that each schema construct, `c`, in any schema along the pathway `S` $\rightarrow$ `GS` has two attributes: *relateTP* is the transformation step that created `c`, and *extent* is the current extent of `c`. If a schema construct remains in the global schema `GS` directly from the source schema `S`, its *relateTP* value is empty.

In our algorithms, each transformation step *tp* has four attributes:

- *action*, which is *"add"*, *"ren"* or *"del"*;

[3]The cases of extend and contract transformations will be considered later in Chapter 6.

- *query*, which is the query used in this transformation step (if any);
- *source*, which for a `renameT(c',c)` returns just `c'`, and for an `addT(c,q)` returns a sequence of all the schema constructs appearing in `q`; and
- *result* which is `c` for both `renameT(c',c)` and `addT(c,q)`.

In case (b) discussed above, where the construct `c` was defined by a transformation step `renameT(c',c)`, the lineage data in `c'` of a bag of tracing tuples T in the extent of `c` is just T itself, and we define this to be both the affect-pool and the origin-pool of T in `c'`.

In case (a), where the construct `c` was created by a transformation step `addT(c,q)`, the key point is how to trace the lineage using the query `q`. We can use the formulae of Theorem 1 to obtain the lineage of data created in this case. The procedures affectPoolOfTuple(t, c) and originPoolOfTuple(t, c) in Figure 5.1 below can be applied to trace the affect pool and origin pool of a tuple t in the extent of schema construct `c`. The result of these procedures, `DL`, is a sequence of pairs

$$\langle\{\mathtt{dl}_1, \mathtt{c}_1\}, \ldots, \{\mathtt{dl}_n, \mathtt{c}_n\}\rangle$$

in which each $\mathtt{dl}_i$ is a bag which contains t's derivation within the extent of schema construct $\mathtt{c}_i$. Note that in these procedures, the sequence $\mathtt{D}^*$ returned by the tracing queries $\mathtt{TQ}^{AP}$ and $\mathtt{TQ}^{OP}$ may consist of bags from different schema constructs. For any such bag, `B`, `B`.*construct* denotes the schema construct from whose extent `B` originates.

Similarly, by Theorem 2, two procedures affectPoolOfSet(T, c) and originPoolOfSet(T, c) can then be used to compute the derivations of a set of tracing tuples T. Since duplicate tuples have an identical derivation, we eliminate any duplicate items and convert the tracing bag to a tracing set first. The procedure

```
proc      affectPoolOfTuple$(t, \mathsf{c})$
input :   a tracing tuple $t$ in the extent of construct c
output :  $t$'s affect-pool, DL
begin
          $\mathtt{D} = [\{\mathtt{O}.\mathit{extent}, \mathtt{O}\} \mid \mathtt{O} \leftarrow \mathsf{c}.\mathit{relateTP.source}]$
          $\mathtt{D}^* = \mathtt{TQ}^{AP}_{\mathtt{D}}(t);$
          $\mathtt{DL} = [\{\mathtt{B}, \mathtt{B}.\mathit{construct}\} \mid \mathtt{B} \leftarrow \mathtt{D}^*]$
          return DL;
end

proc      originPoolOfTuple$(t, \mathsf{c})$
input :   a tracing tuple $t$ in the extent of construct c
output :  $t$'s origin-pool, DL
begin
          $\mathtt{D} = [\{\mathtt{O}.\mathit{extent}, \mathtt{O}\} \mid \mathtt{O} \leftarrow \mathsf{c}.\mathit{relateTP.source}]$
          $\mathtt{D}^* = \mathtt{TQ}^{OP}_{\mathtt{D}}(t);$
          $\mathtt{DL} = [\{\mathtt{B}, \mathtt{B}.\mathit{construct}\} \mid \mathtt{B} \leftarrow \mathtt{D}^*]$
          return DL;
end
```

Figure 5.1: Procedures affectPoolOfTuple and originPoolOfTuple

affectPoolOfSet(T, c) is illustrated in Figure 5.2. The procedure originPoolOfSet(T, c) is identical, with $\mathtt{TQ}_{\mathtt{D}}^{AP}(T)$ replacing $\mathtt{TQ}_{\mathtt{D}}^{OP}(T)$.

```
proc      affectPoolOfSet$(T, \mathsf{c})$
input :   a tracing tuple set $T$ contained in construct $\mathtt{c}$
output :  $T$'s affect-pool, $\mathtt{DL}$
begin
          $\mathtt{D} = [\{\mathtt{O}.\mathit{extent}, \mathtt{O}\} \mid \mathtt{O} \leftarrow \mathsf{c}.\mathit{relateTP}.\mathit{source}]$
          $\mathtt{D}^* = \mathtt{TQ}_{\mathtt{D}}^{AP}(T);$
          $\mathtt{DL} = [\{\mathtt{B}, \mathtt{B}.\mathit{construct}\} \mid \mathtt{B} \leftarrow \mathtt{D}^*]$
          return DL;
end
```

Figure 5.2: Procedure affectPoolOfSet

The algorithms affectPoolOfTuple and affectPoolOfSet, as well as originPoolOfTuple and originPoolOfSet, are correct in the sense that the affect-pool and origin-pool obtained by them conform to the definitions of affect-pool and origin-pool for a SIQL query in Section 5.3. This is because they use the DLT formulae in Section 5.4 to compute the lineage data.

Finally, we give below our algorithm traceAffectPool$(B$, $\mathsf{c})$ in Figure 5.3 for tracing affect lineage using entire transformation pathways given the integrated schema $\mathtt{GS}$, the source schema $\mathtt{S}$, and a transformation pathway $tp_1, \ldots, tp_r$ from $\mathtt{S}$ to $\mathtt{GS}$. Here, B is a bag of tuples contained in the extent of a schema construct $\mathsf{c} \in \mathtt{GS}$. We recall that each schema construct has attributes *relateTP* and *extent*, and that each transformation step has attributes *action*, *query*, *source* and *result*.

The algorithm examines each transformation step from tp_r down to tp_1. If it is a delete step, we ignore it. Otherwise we determine if the *result* of this step is contained in the current $\mathtt{DL}$. If so, we then trace the data lineage of the current data of c in DL, merge the result into DL, and delete c from DL. Because a tuple t^* can be the lineage of both t_i and t_j $(i \neq j)$, if t^* and all of its copies in a data source have already been added to DL as the lineage of t_i, we do not add them again into DL as the lineage of t_j. This is accomplished by the procedure merge given in Figure 5.4 below, where the operator $''-''$ removes an element from a sequence and the operator $''+''$ appends an element to a sequence. At the end of this processing the resulting DL is the lineage of B in the data sources.

The procedure traceAffectPool is correct in the sense that the affect-pool obtained by it conforms to the definitions of affect-pool for a query-sequence in Section 5.5.1. This is because this procedure calls affectPoolOfSet to compute the lineage data based on one add transformation step, and obtains the final lineage data after checking all add transformations along a transformation pathway in reverse.

The exact complexity of the overall DLT process is $O(n \times m)$ where n is the number of add transformations relevant to the tracing data in the transformation pathway and m is the number of different schema constructs in the computed lineage data. By relevant to the tracing data, we mean those transformation steps from the data sources which directly or indirectly create the global schema construct containing the tracing data. The complexity is $O(n \times m)$ because for each add transformation step relevant to the tracing data, the DLT process is performed once for each different schema construct present in the computed lineage data.

We illustrate the use of the traceAffectPool procedure above by means of a simple example. Referring back to the example schema transformation in Section 5.2.3, suppose we have a tracing tuple $t = \{'\texttt{Maths}', 2500\}$ in the extent of $\langle\langle$department, avgDeptSalary$\rangle\rangle$ in GS. The affect-pool, DL, of this tuple is traced as follows.

```
proc      traceAffectPool($B$, c)
input :   tracing tuple bag $B$ contained in construct c;
          transformation pathway $tp_1, \ldots, tp_r$
output :  $B$'s affect-pool, DL
begin
          DL = $\langle\{B, \mathsf{c}\}\rangle$;
          for $j = r$ downto 1 do {
             case ($tp_j$.action = "del")
                continue;
             case ($tp_j$.action = "ren")
                if ($tp_j$.result = $\mathsf{c}_i$ for some $\mathsf{c}_i$ in DL) then
                   DL = (DL $-$ {$\mathtt{dl}_i, \mathsf{c}_i$}) + {$\mathtt{dl}_i, tp_j$.source};
             case ($t_j$.action = "add")
                if ($tp_j$.result = $\mathsf{c}_i$ for some $\mathsf{c}_i$ in DL) then {
                   DL = DL $-$ {$\mathtt{dl}_i, \mathsf{c}_i$};
                   $\mathtt{dl}_i$ = distinct $\mathtt{dl}_i$;
                   DL = merge(DL, affectPoolOfSet($\mathtt{dl}_i, \mathsf{c}_i$)); }
             }
          return DL;
end
```

Figure 5.3: Procedure traceAffectPool

Initially, $\mathtt{DL} = \langle\{\{'\mathtt{Maths}', 2500\}, \langle\langle \mathsf{department}, \mathsf{avgDeptSalary} \rangle\rangle\}\rangle$. traceAffectPool ignores all the delete steps, and finds the add transformation step whose *result* is "$\langle\langle$department, avgDeptSalary$\rangle\rangle$". This is step (7.5), $tp_{(7.5)}$, and:

$tp_{(7.5)}$.*query* = $\langle\langle$avgMathsSalary$\rangle\rangle$ ++ $\langle\langle$avgCompSciSalary$\rangle\rangle$ and

$tp_{(7.5)}$.*source* = [$\langle\langle$avgMathsSalary$\rangle\rangle$, $\langle\langle$avgCompSciSalary$\rangle\rangle$]

Using algorithm affectPoolOfSet, t's lineage at $tp_{(7.5)}$ is as follows:

```
proc      $\mathsf{merge}(\mathtt{DL}, \mathtt{DL}^{new})$
input :   data lineage sequence $\mathtt{DL} = \langle\{\mathtt{dl}_1, \mathtt{c}_1\}, \ldots, \{\mathtt{dl}_n, \mathtt{c}_n\}\rangle$;
          new data lineage sequence$\mathtt{DL}^{new}$
output :  merged data lineage sequence $\mathtt{DL}$
begin
          for each $\{\mathtt{dl}^{new}, \mathtt{c}^{new}\} \in \mathtt{DL}^{new}$ do {
            if ($\mathtt{c}^{new} = \mathtt{c}_i$ for some $\mathtt{c}_i$ in $\mathtt{DL}$)  then {
              $\mathtt{oldData} = \mathtt{dl}_i$;
              $\mathtt{newData} = \mathtt{oldData} ++$
                        $[\mathtt{x} \mid \mathtt{x} \leftarrow \mathtt{dl}^{new}; \mathtt{not}\ (\mathtt{member\ oldData\ x})]$;
              $\mathtt{DL} = (\mathtt{DL} - \{\mathtt{oldData}, \mathtt{c}_i\}) + \{\mathtt{newData}, \mathtt{c}_i\}$;
             }
            else
              $\mathtt{DL} = \mathtt{DL} + \{\mathtt{dl}^{new}, \mathtt{c}^{new}\}$;
           }
          return DL;
end
```

Figure 5.4: Procedure merge

$$\begin{aligned}\mathtt{DL}_{(7.5)} &= \langle\{[\mathtt{x}|\mathtt{x} \leftarrow \langle\!\langle \mathsf{avgMathsSalary} \rangle\!\rangle; \mathtt{x} = \{'\mathtt{Maths}', 2500\}], \langle\!\langle \mathsf{avgMathsSalary} \rangle\!\rangle\}, \\ &\quad\ \{[\mathtt{x}|\mathtt{x} \leftarrow \langle\!\langle \mathsf{avgCompSciSalary} \rangle\!\rangle; \mathtt{x} = \{'\mathtt{Maths}', 2500\}], \langle\!\langle \mathsf{avgCompSciSalary} \rangle\!\rangle\}\rangle \\ &= \langle\{\{'\mathtt{Maths}', 2500\}, \langle\!\langle \mathsf{avgMathsSalary} \rangle\!\rangle\}, \{\emptyset, \langle\!\langle \mathsf{avgCompSciSalary} \rangle\!\rangle\}\rangle \\ &= \langle\{\{'\mathtt{Maths}', 2500\}, \langle\!\langle \mathsf{avgMathsSalary} \rangle\!\rangle\}\rangle\end{aligned}$$

After removing $\{\{'\mathtt{Maths}', 2500\}, \langle\!\langle \mathsf{department}, \mathsf{avgDeptSalary} \rangle\!\rangle\}$, the original tuple, and merging its lineage $\mathtt{DL}_{(7.5)}$, we obtain the updated lineage data as $\langle\{\{'\mathtt{Maths}', 2500\}, \langle\!\langle \mathsf{avgMathsSalary} \rangle\!\rangle\}\rangle$. Similarly, we obtain the data lineage relating to this

DL. Thus, $\mathtt{DL}_{(7.2)}$ is all of the tuples in $\langle\langle\mathsf{mathsSalary}\rangle\rangle$ and $\mathtt{DL}_{(7.1)}$ is all of the tuples in $\langle\langle\mathsf{mathematician},\mathsf{salary}\rangle\rangle$, where construct $\langle\langle\mathsf{mathematician},\mathsf{salary}\rangle\rangle$ is a base collection in SS.

We conclude that the affect-pool of tuple $\{'\mathtt{Maths}', 2500\}$ in the extent of $\langle\langle\mathsf{department},\mathsf{avgDeptSalary}\rangle\rangle$ in GS consists of all of the tuples in the extent of $\langle\langle\mathsf{mathematician},\mathsf{salary}\rangle\rangle$ in SS.

Procedure traceOriginPool(B, c) is similar, obtained by replacing affectPoolOfSet by originPoolOfSet.

Note that we have not implemented these DLT algorithms which assume fully materialised transformation pathways. In Chapter 6, we develop a generalised DLT algorithm for general transformation pathways where intermediate schema constructs may or may not be materialised. The implementation of this generalised DLT algorithm is discussed in Appendix C.

5.6 IQL^c to SIQL Decomposition Order

With the decomposition rules described in 5.2.2, we decompose a general IQL^c query into a sequence of SIQL queries by means of a depth-first traversal of the IQL^c query tree. However, does the traversal order affect the process of tracing data lineage, *i.e.* would we get the same lineage data irrespective of the order of decomposition? In this section, we investigate the problem of decomposition order and conclude that the order of traversing an IQL^c query tree does not affect the result of our DLT process.

Firstly, if an IQL^c query is a list of constants, *i.e.* $[\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_n]$, there is no traversal order problem. We next discuss the situation of a query having arguments.

If a query is an 1-argument IQL^c query, just one order of traversing the query is available. If the one sub-query has no traversal order problem, the main query will not have the traversal order problem.

If a query is a 2-argument IQL^c query, *i.e.* `E1 -- E2`, there may be two sub-queries in the main query and two orders of traversing the query, *e.g.*

```
v1  =  E1
v2  =  E2
v   =  v1 -- v2
```

and

```
v1  =  E2
v2  =  E1
v   =  v2 -- v1
```

However, there is no traversal order problem since the DLT formulae for each data source in the $--$ expression are independent of each other.

If a query is a n-argument IQL^c query, such as an ++ expression, since the places of its arguments are exchangeable, there are various orders of traversing the query. However, again the DLT formulae for each data source in a ++ expression are independent of each other. Thus, the order of traversal does not affect the result of tracing lineage data in the data sources.

Otherwise, if the n-argument IQL^c query is a comprehension, we consider the following three cases.

- One, the comprehension is not a select-join comprehension and has to be transformed into a select-join comprehension. There is no traversal order problem in this transformation since we use a `map` expression to achieve this transformation.

- Two, the select-join comprehension does not contain `member` filters. In this case, similar to the situation of ++ expressions, the DLT formulae for each data source in the comprehension are independent of each other and there is no traversal order problem;

- Three, the select-join comprehension contains member filters. On the one hand, if the select-join comprehension contains just one generator and member filter, similar to the situation of $--$ expressions, the DLT formulae for each data source are independent of each other and there is no traversal problem.

  On the other hand, if the select-join comprehension contains multiple generators and member filters, $[\mathtt{p}|\mathtt{G1};\mathtt{G2};...;\mathtt{Gr};\mathtt{M1};...;\mathtt{Ms};\mathtt{C1};\ldots;\mathtt{Ct}]$, according to the decomposition rules in Section 5.2.2, this comprehension is decomposed into following SIQL comprehensions:

  $$\begin{aligned}
  \mathtt{v}_1 &= [\mathtt{p}|\mathtt{G1};\ldots;\mathtt{Gr};\mathtt{C1};...;\mathtt{Ct}] \\
  \mathtt{v}_2 &= [\mathtt{p}|\mathtt{p} \leftarrow \mathtt{v}_1;\mathtt{M1}] \\
  \mathtt{v}_3 &= [\mathtt{p}|\mathtt{p} \leftarrow \mathtt{v}_2;\mathtt{M2}] \\
  \ldots & \\
  \mathtt{v}_s &= [\mathtt{p}|\mathtt{p} \leftarrow \mathtt{v}_{s-1};\mathtt{M}_{s-1}] \\
  \mathtt{v} &= [\mathtt{p}|\mathtt{p} \leftarrow \mathtt{v}_s;\mathtt{M}_s]
  \end{aligned}$$

  Although the order of traversing the member filters such as $\mathtt{M1},...,\mathtt{Ms}$ could be changed, according to the DLT formulae in Section 5.4, the obtained lineage data in all the intermediate views $\mathtt{v}_1,...,\mathtt{v}_s$ is the tracing tuple t itself while the obtained lineage data for each member filter $\mathtt{M}_i$ is a lambda expression over the tracing tuple t. Both of these cannot be affected by the traversal order. Furthermore, each individual view $\mathtt{v},\mathtt{v}_1,...,\mathtt{v}_s$ is a select-join comprehension either with only one generator and member filter, or without any member filters, and which therefore has no traversal order problem.

In summary, the order of traversing an IQL^c query tree does not affect the result of our DLT process.

5.7 Ambiguity of Lineage Data

The ambiguity of lineage data, also called *derivation inequivalence* [CWW00], relates to the fact that for queries which are equivalent but different syntactically DLT processes may obtain different lineage data for identical tracing data. This section investigates how this problem may happen in our context. Two queries are *equivalent* if they give identical results for all possible values of their base collections. That is, given two queries q_1 and q_2 both referring to base collections $\mathtt{b}_1, ..., \mathtt{b}_n$, q_1 and q_2 are equivalent if $q_1[\mathtt{b}_1/\mathtt{I}_1, ..., \mathtt{b}_n/\mathtt{I}_n] = q_2[\mathtt{b}_1/\mathtt{I}_1, ..., \mathtt{b}_n/\mathtt{I}_n]$ is true for all instances $\mathtt{I}_1, ..., \mathtt{I}_n$ of $\mathtt{b}_1, ..., \mathtt{b}_n$ respectively. We use $\mathtt{v1} \equiv \mathtt{v2}$ to denote that views `v1` and `v2` are defined by equivalent queries.

5.7.1 Derivation for difference and not member Operations

Ambiguity of lineage data may happen when difference (*i.e.* $--$ in IQL^c) and not member operations are involved in the view definitions.

For example, consider two bags $\mathtt{R} = [0, 1, 1, 2, 3]$, $\mathtt{S} = [-1, 1, 2, 3, 3]$. Two pairs of equivalent views, $\mathtt{v1} \equiv \mathtt{v2}$ and $\mathtt{v3} \equiv \mathtt{v4}$, are defined as follows.

$$
\begin{aligned}
\mathtt{v1} &= \mathtt{R} -- (\mathtt{R} -- \mathtt{S}) = [1, 2, 3] \\
\mathtt{v2} &= \mathtt{S} -- (\mathtt{S} -- \mathtt{R}) = [1, 2, 3] \\
\mathtt{v3} &= [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{R}; \mathtt{member\ S\ x}] = [1, 1, 2, 3] \\
\mathtt{v4} &= [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{R}; \mathtt{not\ (member\ } [\mathtt{y}|\mathtt{y} \leftarrow \mathtt{R}; \mathtt{not\ (member\ S\ y)}]\ \mathtt{x})] = [1, 1, 2, 3]
\end{aligned}
$$

The lineage of data in an IQL^c view can be traced by decomposing the view into a sequence of intermediate SIQL views. In order to trace the lineage of data in the above four views, intermediate views are required as follows:

For $\texttt{v1}$, $\texttt{v1'} = (\texttt{R} -- \texttt{S}) = [0,1]$.

For $\texttt{v2}$, $\texttt{v2'} = (\texttt{S} -- \texttt{R}) = [-1,3]$.

For $\texttt{v3}$, no intermediate view needed.

For $\texttt{v4}$, $\texttt{v4'} = [\texttt{y|y} \leftarrow \texttt{R}; \texttt{not}\ (\texttt{member S y})] = [0]$.

With the above intermediate views, we can now trace the lineage of the views' data. For example, the affect-pool of the data item $t = 1 \in \texttt{v1}$ and $t = 1 \in \texttt{v2}$ are as follows. Here, we denote by $\texttt{D|dl}$ the lineage data $\texttt{dl}$ in the collection $\texttt{D}$, *i.e.* all instances of the tuple $\texttt{dl}$ in the bag $\texttt{D}$ (the result of the query $[x|x \leftarrow \texttt{D}; x = \texttt{dl}]$).

$$
\begin{aligned}
\texttt{AP}_{\texttt{v1}}(t) &\overset{t2}{=} \langle \texttt{R}|[\texttt{x|x} \leftarrow \texttt{R}; \texttt{x} = 1], \texttt{R} -- \texttt{S}\rangle \\
&= \langle \texttt{R}|[1,1], \texttt{v1'}\rangle \\
&\overset{T2}{=} \langle \texttt{R}|[1,1], \texttt{R}|[\texttt{x|x} \leftarrow \texttt{R}; \texttt{member v1' x}], \texttt{S}\rangle \\
&= \langle \texttt{R}|[1,1], \texttt{R}|[\texttt{x|x} \leftarrow \texttt{R}; \texttt{member}\ [0,1]\ \texttt{x}], \texttt{S}\rangle \\
&= \langle \texttt{R}|[1,1], \texttt{R}|[0,1,1], \texttt{S}|[-1,1,2,3,3]\rangle \\
&= \langle \texttt{R}|[0,1,1], \texttt{S}|[-1,1,2,3,3]\rangle \\
\texttt{AP}_{\texttt{v2}}(t) &\overset{t2}{=} \langle \texttt{S}|[\texttt{x|x} \leftarrow \texttt{S}; \texttt{x} = 1], \texttt{S} -- \texttt{R}\rangle \\
&= \langle \texttt{S}|[1], \texttt{v2'}\rangle \\
&\overset{T2}{=} \langle \texttt{S}|[1], \texttt{S}|[\texttt{x|x} \leftarrow \texttt{S}; \texttt{member v2' x}], \texttt{R}\rangle \\
&= \langle \texttt{S}|[1], \texttt{S}|[\texttt{x|x} \leftarrow \texttt{S}; \texttt{member}\ [-1,3]\ \texttt{x}], \texttt{R}\rangle \\
&= \langle \texttt{S}|[1], \texttt{S}|[-1,3,3], \texttt{R}|[0,1,1,2,3]\rangle \\
&= \langle \texttt{R}|[0,1,1,2,3], \texttt{S}|[-1,1,3,3]\rangle
\end{aligned}
$$

We can see that the affect-pool of identical tracing data in $\texttt{v1}$ and $\texttt{v2}$ are inequivalent. The affect-pool of tuple $t = 1 \in \texttt{v3}$ and $t = 1 \in \texttt{v4}$ are:

$$\begin{aligned}
\mathtt{AP}_{\mathtt{v3}}(t) &\overset{t12}{=} \langle \mathtt{R}|[\mathtt{x}\,|\,\mathtt{x} \leftarrow \mathtt{R}; \mathtt{x} = 1], \mathtt{S}|[\mathtt{x}\,|\,\mathtt{x} \leftarrow \mathtt{S}; \mathtt{x} = 1]\rangle \\
&= \langle \mathtt{R}|[1,1], \mathtt{S}|[1]\rangle \\
\mathtt{AP}_{\mathtt{v4}}(t) &\overset{t13}{=} \langle \mathtt{R}|[\mathtt{x}\,|\,\mathtt{x} \leftarrow \mathtt{R}; \mathtt{x} = 1], \mathtt{v4'}\rangle \\
&\overset{T13}{=} \langle \mathtt{R}|[1,1], \mathtt{R}|[\mathtt{y}\,|\,\mathtt{y} \leftarrow \mathtt{R}; \mathtt{member\ v4'\ y}], \mathtt{S}\rangle \\
&= \langle \mathtt{R}|[1,1], \mathtt{R}|[\mathtt{y}\,|\,\mathtt{y} \leftarrow \mathtt{R}; \mathtt{member}\ [0]\ \mathtt{y}], \mathtt{S}\rangle \\
&= \langle \mathtt{R}|[1,1], \mathtt{R}|[0], \mathtt{S}|[-1,1,2,3,3]\rangle \\
&= \langle \mathtt{R}|[0,1,1], \mathtt{S}|[-1,1,2,3,3]\rangle
\end{aligned}$$

We can see that the affect-pool of above identical tracing data in v3 and v4 are also inequivalent.

The reason for the inequivalent affect-pool of the data in views defined by equivalent queries involving the $--$ and not member operators is the definition of affect-pool. As described in Section 5.4, the affect-pool in a data source D2 in queries of the form D1 $--$ D2 or $[x|x \leftarrow \mathtt{D1}; \mathtt{not}\ (\mathtt{member\ D2}\ x)]$, includes all data in D2. So the computed affect-pool in D2 may contain some "irrelevant" data which does not affect the existence of the tracing data in the view.

For example, if the tracing data is $t = 1$ in the view R $--$ S, the irrelevant data in S are $[-1, 2, 3, 3]$, which are also included in t's affect-pool.

Although origin-pool is defined to contain the minimal essential lineage data in a data source, ambiguity of lineage data may also occur for tracing origin-pool. For example, in the case of the above four views, the origin-pool of the tracing data item $t = 1$ are also inequivalent (we use $\mathtt{D}|\emptyset$ to denote no lineage data in D):

$$\begin{aligned}
\mathtt{OP}_{\mathtt{v1}}(t) &\overset{t2}{=} \langle \mathtt{R}|[\mathtt{x}\,|\,\mathtt{x} \leftarrow \mathtt{R}; \mathtt{x} = 1], (\mathtt{R} -- \mathtt{S})|[\mathtt{x}\,|\,\mathtt{x} \leftarrow (\mathtt{R} -- \mathtt{S}); \mathtt{x} = 1]\rangle \\
&= \langle \mathtt{R}|[1,1], \mathtt{v1'}|[\mathtt{x}\,|\,\mathtt{x} \leftarrow [0,1]; \mathtt{x} = 1]\rangle \\
&= \langle \mathtt{R}|[1,1], \mathtt{v1'}|[1]\rangle \\
&\overset{t2}{=} \langle \mathtt{R}|[1,1], \mathtt{R}|[\mathtt{x}\,|\,\mathtt{x} \leftarrow \mathtt{R}; \mathtt{x} = 1], \mathtt{S}|[\mathtt{x}\,|\,\mathtt{x} \leftarrow \mathtt{S}; \mathtt{x} = 1]\rangle \\
&= \langle \mathtt{R}|[1,1], \mathtt{R}|[1,1], \mathtt{S}|[1]\rangle \\
&= \langle \mathtt{R}|[1,1], \mathtt{S}|[1]\rangle
\end{aligned}$$

$$\begin{aligned}
\texttt{OP}_{\texttt{v2}}(t) &\overset{t2}{=} \langle \texttt{S}|[\texttt{x}\,|\,\texttt{x} \leftarrow \texttt{S}; \texttt{x}=1], (\texttt{R} -- \texttt{S})|[\texttt{x}\,|\,\texttt{x} \leftarrow (\texttt{S} -- \texttt{R}); \texttt{x}=1]\rangle \\
&= \langle \texttt{S}|[1], \texttt{v2'}|[\texttt{x}\,|\,\texttt{x} \leftarrow [-1,3]; \texttt{x}=1]\rangle \\
&= \langle \texttt{S}|[1], \texttt{v2'}|\emptyset\rangle \\
&= \langle \texttt{S}|[1]\rangle
\end{aligned}$$

and

$$\begin{aligned}
\texttt{OP}_{\texttt{v3}}(t) &\overset{t12}{=} \langle \texttt{R}|[\texttt{x}\,|\,\texttt{x} \leftarrow \texttt{R}; \texttt{x}=1], \texttt{S}|[\texttt{x}\,|\,\texttt{x} \leftarrow \texttt{S}; \texttt{x}=1]\rangle \\
&= \langle \texttt{R}|[1,1], \texttt{S}|[1]\rangle \\
\texttt{OP}_{\texttt{v4}}(t) &\overset{t13}{=} \langle \texttt{R}|[\texttt{x}\,|\,\texttt{x} \leftarrow \texttt{R}; \texttt{x}=1], \texttt{v4'}|\emptyset\rangle \\
&= \langle \texttt{R}|[1,1]\rangle
\end{aligned}$$

5.7.2 Derivation for Aggregate Functions

Ambiguity of lineage data may also happen when queries involve aggregate functions. Suppose that bags R and S are the same as in Section 5.7.1. Consider DLT processes over the following two pairs of equivalent views, $\texttt{v5} \equiv \texttt{v6}$ and $\texttt{v7} \equiv \texttt{v8}$:

$$\begin{aligned}
\texttt{v5} &= \texttt{sum R} = 7 \\
\texttt{v6} &= \texttt{sum } [\texttt{x}\,|\,\texttt{x} \leftarrow \texttt{R}; \texttt{x} \neq 0] = 7 \\
\texttt{v7} &= \texttt{max S} = [3,3] \\
\texttt{v8} &= \texttt{max } [\texttt{x}\,|\,\texttt{x} \leftarrow \texttt{S}; \texttt{x} > (\texttt{min S})] = [3,3]
\end{aligned}$$

The affect-pool of $t = 7 \in \texttt{v5}$ and $t = 7 \in \texttt{v6}$ are:

$$\begin{aligned}
\texttt{AP}_{\texttt{v5}}(t) &\overset{t6}{=} \langle \texttt{R}\rangle &&= \langle \texttt{R}|[0,1,1,2,3]\rangle \\
\texttt{AP}_{\texttt{v6}}(t) &\overset{t6}{=} \langle \texttt{R}|[\texttt{x}\,|\,\texttt{x} \leftarrow \texttt{R}; \texttt{x} \neq 0]\rangle &&= \langle \texttt{R}|[1,1,2,3]\rangle
\end{aligned}$$

and the affect-pool of $t = 3 \in \texttt{v7}$ and $t = 3 \in \texttt{v8}$ are:

$$\begin{aligned}
\texttt{AP}_{\texttt{v7}}(t) &\overset{t6}{=} \langle \texttt{S}\rangle &&= \langle \texttt{S}|[-1,1,2,3,3]\rangle \\
\texttt{AP}_{\texttt{v8}}(t) &\overset{t6}{=} \langle \texttt{S}|[\texttt{x}\,|\,\texttt{x} \leftarrow \texttt{S}; \texttt{x} > (\texttt{min S})]\rangle &&= \langle \texttt{S}|[1,2,3,3]\rangle
\end{aligned}$$

We can see that the affect-pool of identical tracing data for these equivalent views are inequivalent.

The reason for this ambiguity of affect-pool is that, according to the DLT formulae of affect-pool in Section 5.4, the affect-pool of data in an aggregate view includes all the data in the data source, which can bring irrelevant data into the derivation. In above example, views `v6` and `v8` filter off some irrelevant data by using predicate expressions, so that the computed affect-pool over the two views does not contain this irrelevant data.

Such problems may be avoided in tracing the origin-pool, since the origin-pool is defined to contain the minimal essential lineage data in the data sources, and any data item and its duplicates in the origin-pool are non-redundant.

For example, the origin-pool of $t = 7 \in \texttt{v5}$ and $t = 7 \in \texttt{v6}$ are identical:

$$\begin{aligned} \texttt{OP}_{\texttt{v5}}(t) &\overset{t6}{=} \langle \texttt{R}|[\texttt{x|x} \leftarrow \texttt{R}; \texttt{x} \neq 0]\rangle &= \langle \texttt{R}|[1,1,2,3]\rangle \\ \texttt{OP}_{\texttt{v6}}(t) &\overset{t6}{=} \langle \texttt{R}|[\texttt{x|x} \leftarrow [\texttt{y|y} \leftarrow \texttt{R}; \texttt{y} \neq 0]; \texttt{x} \neq 0]\rangle &= \langle \texttt{R}|[1,1,2,3]\rangle \end{aligned}$$

and the origin-pool of $t = 3 \in \texttt{v7}$ and $t = 3 \in \texttt{v8}$ are also identical:

$$\begin{aligned} \texttt{OP}_{\texttt{v7}}(t) &\overset{t5}{=} \langle \texttt{S}|[\texttt{x|x} \leftarrow \texttt{S}; \texttt{x} = 3]\rangle &= \langle \texttt{S}|[3,3]\rangle \\ \texttt{OP}_{\texttt{v8}}(t) &\overset{t5}{=} \langle \texttt{S}|[\texttt{x|x} \leftarrow [\texttt{y|y} \leftarrow \texttt{S}; \texttt{y} > (\texttt{min S})]; \texttt{x} = 3]\rangle &= \langle \texttt{S}|[3,3]\rangle \end{aligned}$$

However, the derivation inequivalence problem cannot always be avoided in tracing the origin-pool. For example, suppose two equivalent views $\texttt{v9} \equiv \texttt{v10}$ are defined as follows:

$$\begin{aligned} \texttt{v9} &= \texttt{sum S} = 8 \\ \texttt{v10} &= \texttt{sum } [\texttt{x|x} \leftarrow \texttt{S}; \texttt{not } (\texttt{member } [\texttt{x1|x1} \leftarrow \texttt{S}; \texttt{x2} \leftarrow \texttt{S}; \texttt{x1} = (-\texttt{x2})]\ \texttt{x})] = 8 \end{aligned}$$

In order to trace the origin-pool of `v10`'s data, the intermediate views for `v10` are defined as follows:

$$\begin{aligned} \texttt{v10'} &= [\texttt{x1|x1} \leftarrow \texttt{S}; \texttt{x2} \leftarrow \texttt{S}; \texttt{x1} = (-\texttt{x2})] = [-1, 1] \\ \texttt{v10''} &= [\texttt{x|x} \leftarrow \texttt{S}; \texttt{not } (\texttt{member v10' x})] = [2,3,3] \\ \texttt{v10} &= \texttt{sum v10''} = 8 \end{aligned}$$

Then, the origin-pool of $t = 8 \in \texttt{v9}$ and $t = 8 \in \texttt{v10}$ are:

$$\begin{aligned}
\mathtt{OP}_{\mathtt{v9}}(t) &\overset{t6}{=} \langle \mathtt{S}|[\mathtt{x}|\mathtt{x} \leftarrow \mathtt{S}; \mathtt{x} \neq 0]\rangle \\
&= \langle \mathtt{S}|[-1,1,2,3,3]\rangle \\
\mathtt{OP}_{\mathtt{v10}}(t) &\overset{t6}{=} \langle \mathtt{v10''}|[\mathtt{x}|\mathtt{x} \leftarrow \mathtt{v10''}; \mathtt{x} \neq 0]\rangle = \langle \mathtt{v10''}|[2,3,3]\rangle \\
&= \langle [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{S}; \mathtt{not\ (member\ v10'\ x)}]|[2,3,3]\rangle \\
&\overset{T11}{=} \langle \mathtt{S}|[2,3,3], \mathtt{v10'}|\emptyset\rangle \\
&= \langle \mathtt{S}|[2,3,3]\rangle
\end{aligned}$$

We can see that $\mathtt{OP}_{\mathtt{v9}}(t) \neq \mathtt{OP}_{\mathtt{v10}}(t)$. This is because the view `v10` is firstly applying a select operation over the data source `S`, to eliminate data item d in `S` and its inverse d^{-1}, *i.e.* $d + d^{-1} = 0$.

5.7.3 Derivation for Where-Provenance

The problem of where-provenance is introduced in Buneman *et al.*'s work [BKT01]. In that paper, tracing the where-provenance of a tracing tuple consists of finding the lineage of one component of the tuple, rather than the whole tuple. Also, the where-provenance is not exact data, but rather a path for describing where the lineage is. That paper describes that derivation inequivalence may happen when tracing where-provenance.

Examples of where-provenance inequivalence [4]

Suppose that `w1` is a view over a relational table $\langle\langle\mathsf{Employee}\rangle\rangle$, where the extent of $\langle\langle\mathsf{Employee}\rangle\rangle$ table is a list of 3-item tuples containing `name`, `salary` and `bonus` information of employees. The definition of `w1` is as follows:

$\mathtt{w1} = [\{\mathtt{name,salary}\}|\{\mathtt{name,salary,bonus}\} \leftarrow \langle\langle\mathsf{Employee}\rangle\rangle; \mathtt{salary} = 1200]$

If $\{'\mathtt{Tom}', 1200\}$ is a tuple in `w1` and the data 1200 in the tuple only comes from the tuple $\{'\mathtt{Tom}', 1200, 1000\}$ in the extent of $\langle\langle\mathsf{Employee}\rangle\rangle$, then the where-provenance of 1200 is the path $''\langle\langle\mathsf{Employee}\rangle\rangle.\{\mathtt{name}: '\mathtt{Tom}'\}.\mathtt{salary}''$, which means that 1200

[4]The examples illustrated in this section are derived from [BKT01].

comes from the attribute salary in the relation ⟨⟨Employee⟩⟩ where the value of the attribute name is ′Tom′.

However, if we consider the following view w2 over construct ⟨⟨Employee⟩⟩, which is an equivalent view to w1,

$$\texttt{w2} \;=\; [\{\texttt{name}, 1200\}|\{\texttt{name,salary,bonus}\} \leftarrow \langle\langle \mathsf{Employee}\rangle\rangle; \texttt{salary} = 1200]$$

the where-provenance of 1200 in $\{'\texttt{Tom}', 1200\}$ is the query (view definition) itself, since the value is directly appearing in the query expression.

Another example illustrating inequivalent where-provenance is as follows. Suppose that $\texttt{w3} \equiv \texttt{w4}$ where

$$\begin{aligned}\texttt{w3} &= [\{\texttt{id,ns}\}|\{\texttt{id,s,b,ns}\} \leftarrow \langle\langle \mathsf{D}\rangle\rangle; \texttt{s = b}; \texttt{s = ns}]\\ \texttt{w4} &= [\{\texttt{id,ns}\}|\{\texttt{id,s,b,ns}\} \leftarrow \langle\langle \mathsf{D}\rangle\rangle;\\ &\qquad\qquad \texttt{member } [\{\texttt{id1,ns1}\}|\{\texttt{id1,s1,b1,ns1}\} \leftarrow \langle\langle \mathsf{D}\rangle\rangle; \texttt{s1 = b1}]\ \{\texttt{id,ns}\};\\ &\qquad\qquad \texttt{s = ns}]\end{aligned}$$

In the case of w3, the attribute ns in the result view depends on attributes: s, b and ns, in relational table ⟨⟨D⟩⟩. While in the case of w4, the attribute ns in the result view depends on attributes: id, s, b and ns, in ⟨⟨D⟩⟩.

In our DLT approach, we only consider tracing the lineage data of an entire tuple, which is termed why-provenance in [BKT01]. However, in AutoMed, each extensional modelling construct of a high-level modelling language is specified as an HDM node or edge and cannot be broken down further. For example, each attribute in a relational table is a construct in the AutoMed relational schema.

In other words, in our DLT approach, not only the why-provenance but also the where-provenance has been considered, when the AutoMed data modelling technique is used for modelling data, *e.g.*, using the simple relational data model. In this sense, we deal with the problem of tracing where-provenance and why-provenance simultaneously, so that the problem of inequivalent where-provenance is avoided.

For example, by using the simple relational data model and SIQL queries, the above four view definitions can be rewritten (denoted as $\leadsto$) as follows. In the simple relational data model, constructs of the relational table $\langle\langle\mathsf{Employee}\rangle\rangle$ include: $\langle\langle\mathsf{Employee}\rangle\rangle$, $\langle\langle\mathsf{Employee,name}\rangle\rangle$, $\langle\langle\mathsf{Employee,salary}\rangle\rangle$ and $\langle\langle\mathsf{Employee,bonus}\rangle\rangle$; constructs of the table $\langle\langle\mathsf{D}\rangle\rangle$ include: $\langle\langle\mathsf{D}\rangle\rangle$, $\langle\langle\mathsf{D,id}\rangle\rangle$, $\langle\langle\mathsf{D,s,}\rangle\rangle$, $\langle\langle\mathsf{D,b}\rangle\rangle$ and $\langle\langle\mathsf{D,ns}\rangle\rangle$.

$$\begin{aligned}
\texttt{w1} \leadsto\ \texttt{w1'} &= [\texttt{\{name,salary\}}\,|\,\texttt{\{name,salary\}} \leftarrow \langle\langle\mathsf{Employee,salary}\rangle\rangle;\\
&\qquad\qquad\qquad \texttt{salary} = 1200]\\
\texttt{w2} \leadsto\ \texttt{w2'} &= [\texttt{\{name,salary\}}\,|\,\texttt{\{name,salary\}} \leftarrow \langle\langle\mathsf{Employee,salary}\rangle\rangle;\\
&\qquad\qquad\qquad \texttt{salary} = 1200]\\
\texttt{w2''} &= \texttt{map}\ (\texttt{lambda \{name,salary\}}.\{\texttt{name}, 1200\})\ \texttt{w2'}
\end{aligned}$$

Obviously, `w1'` and `w2'` are identical, and `w2''` uses a `lambda` expression replacing by the constant 1200 the salary values in the result of `w2'`. Here, we cannot trace the lineage data of 1200 separately. If it is required to do that, definitions of `w1` and `w2` can be rewritten as:

$$\begin{aligned}
\texttt{w1} \leadsto\ \texttt{w1a'} &= [\texttt{\{name,salary\}}\,|\,\texttt{\{name,salary\}} \leftarrow \langle\langle\mathsf{Employee,salary}\rangle\rangle;\\
&\qquad\qquad\qquad \texttt{salary} = 1200]\\
\texttt{w1a''} &= \texttt{map}\ (\texttt{lambda \{name,salary\}.\{salary\}})\ \texttt{w1a'}\\
\texttt{w2} \leadsto\ \texttt{w2a'} &= [\texttt{\{name,salary\}}\,|\,\texttt{\{name,salary\}} \leftarrow \langle\langle\mathsf{Employee,salary}\rangle\rangle;\\
&\qquad\qquad\qquad \texttt{salary} = 1200]\\
\texttt{w2a''} &= \texttt{map}\ (\texttt{lambda \{name,salary\}}.\{1200\})\ \texttt{w2a'}
\end{aligned}$$

We can see that, although intermediate views `w1a''` and `w2a''` have the same result in the current specific situation, they have different definitions. In this sense, views `w1` and `w2` can be regarded as inequivalent and the problem of derivation inequivalence does not arise for these two views. However, even we admit that these two views are equivalent in the current situation, according to the DLT formula $t15$ in Theorem 1, the lineage data of 1200 in `w1a''` and `w2a''` are obtained as follows:

$$\texttt{w1a'}|[\texttt{\{name,salary\}}|\texttt{\{name,salary\}} \leftarrow \texttt{w1a'}; \texttt{salary} = 1200]$$

$$\texttt{w2a'}|[\texttt{\{name,salary\}}|\texttt{\{name,salary\}} \leftarrow \texttt{w2a'}; 1200 = 1200]$$

Since views `w1a'` and `w2a'` are identical, 1200 over the two views have the same lineage.

As to views `w3` and `w4`, their definitions can be rewritten as follows:

$$\begin{array}{llcl}
\texttt{w3} \leadsto & \texttt{w3'} & = & [\texttt{\{id,s\}}|\texttt{\{id,s\}} \leftarrow \langle\langle \mathsf{D},\mathsf{s}\rangle\rangle; \texttt{member}\ \langle\langle \mathsf{D},\mathsf{b}\rangle\rangle\ \texttt{\{id,s\}}] \\
 & \texttt{w3''} & = & [\texttt{\{id,ns\}}|\texttt{\{id,ns\}} \leftarrow \langle\langle \mathsf{D},\mathsf{ns}\rangle\rangle; \texttt{member}\ \texttt{w3'}\ \texttt{\{id,ns\}}] \\
\texttt{w4} \leadsto & \texttt{w4'} & = & [\texttt{\{id,ns\}}|\texttt{\{id,ns\}} \leftarrow \langle\langle \mathsf{D},\mathsf{ns}\rangle\rangle; \texttt{member}\ \langle\langle \mathsf{D},\mathsf{s}\rangle\rangle\ \texttt{\{id,ns\}}] \\
 & \texttt{w4''} & = & [\texttt{\{id,s\}}\ |\texttt{\{id,s\}} \leftarrow \langle\langle \mathsf{D},\mathsf{s}\rangle\rangle; \texttt{member}\ \langle\langle \mathsf{D},\mathsf{b}\rangle\rangle\ \texttt{\{id,s\}}] \\
 & \texttt{w4'''} & = & [\texttt{\{id,ns\}}|\texttt{\{id,ns\}} \leftarrow \texttt{w4'}; \texttt{member}\ \texttt{w4''}\ \texttt{\{id,ns\}}]
\end{array}$$

We can see that tuple `{id,ns}` in the two views have the same lineage coming from $\langle\langle \mathsf{D},\mathsf{ns}\rangle\rangle$, $\langle\langle \mathsf{D},\mathsf{s}\rangle\rangle$ and $\langle\langle \mathsf{D},\mathsf{b}\rangle\rangle$ constructs.

5.7.4 Summary

This section has investigated when ambiguity of lineage data may happen in our context — the problem may happen when tracing the lineage of the data in views defined by IQL^c queries involving $--$, not member filters and aggregation operations. In Cui *et al*'s work [CWW00], the definition of data lineage results in the same problem of derivation inequivalence.

Ambiguity of lineage may also happen when tracing where-provenance. This section has described how our DLT approach for tracing why-provenance can also be used for tracing where-provenance, so as to reduce the chance of where-provenance inequivalence occurring.

5.8 Discussion

This chapter has given the definitions of data lineage in the context of AutoMed, which we have termed *affect-pool* and *origin-pool*. The affect-pool includes all of the source data that had some influence on the tracing data, while the origin-pool is the specific data in the data sources from which the tracing data is extracted.

We have introduced a subset of the full IQL query language, IQL^c, which incorporates the major relational and aggregation operators on collections; and have used a subset of IQL^c, SIQL, for our data lineage tracing algorithms. Any IQL^c query can be decomposed into a series of transformations with SIQL queries on intermediate schema constructs. We have also discussed that the order of traversing and decomposing an IQL^c query does not affect the result of our DLT process.

DLT formulae for SIQL queries and an algorithm for tracing data lineage over AutoMed transformation pathways have also been presented in this chapter. A limitation of this algorithm is that transformation pathways need to be fully materialised, *i.e.* all the constructs defined by `add` transformations need to be materialised. In the next chapter, we will present a method for tracing data lineage over general AutoMed transformation pathways where intermediate schema constructs may or may not be materialised.

In Section 5.7, we have discussed the ambiguity of lineage data. For identical tracing data based on equivalent queries, inequivalent lineage data may be obtained if the queries involve $--$, `not member` or aggregation operations. Inequivalent lineage data may also be obtained when tracing where-provenance. We observed that the process of tracing where-provenance can be handled by the process of tracing why-provenance when AutoMed is used for modelling data, so that the problem of inequivalent where-provenance can be reduced.

Chapter 6

Generalising the Data Lineage Tracing Algorithm

6.1 Motivation

In Chapter 5 we discussed how to trace the lineage of data in the global database by applying the DLT formulae for SIQL queries to each transformation step in the transformation pathway from the data source schemas to the global schema in reverse, finally ending up with the lineage data in the original data sources.

However, in general transformation pathways not all schema constructs created by `add` transformations will be materialised, and the above simple DLT approach is no longer applicable. In practice, transformation pathways may be virtual or partially materialised, in which intermediate schema constructs may or may not be materialised. Moreover, as described as in Section 4.2.2, a general IQL^c query is decomposed into a sequence of SIQL queries with some new intermediate constructs, and it should not be necessary to materialise these constructs in order to apply DLT.

In this chapter, we assume that a schema transformation pathway may contain virtual intermediate constructs, but that all queries appearing within it are SIQL queries. The DLT algorithm described in Chapter 5 cannot handle virtual intermediate views and so cannot be applied in this situation.

One approach to solving the problem of virtual schema constructs would be to use AutoMed's Global Query Processor to evaluate the query creating the virtual construct and compute its extent, so that the DLT approach of Chapter 5 could be applied. However, this approach is impractical due to the space and time overheads it incurs.

Instead, our approach for handling the problem of virtual schema constructs is that we use a data structure described in Section 6.2, Lineage, to denote lineage data in a schema construct. If the construct is materialised, Lineage contains the actual lineage data. If the construct is virtual, Lineage contains relevant information for deriving the lineage data from the virtual construct. Rather than materialising the virtual construct, we use such virtual lineage data as the tracing data for earlier transformation steps. Repeating this process, finally if the data sources of a transformation step are all materialised, we can obtain the materialised lineage data from these data sources.

In the rest of this chapter, Section 6.2 describes the data structures used by our DLT algorithm. Section 6.3 presents our DLT procedure for a single transformation step. DLT formulae for handling virtual intermediate constructs and lineage data are developed in Section 6.4. Section 6.5 presents DLT algorithms for tracing data lineage along a general transformation pathway. Section 6.6 discusses the usage of queries beyond IQL^c, and of delete, contract and extend transformation steps for DLT. Section 6.7 discusses the implementation of our DLT algorithms described in this chapter. Finally, Section 6.8 summarises and discusses our DLT approach.

6.2 Data Structures for Data Lineage Tracing

In order to handle virtual intermediate lineage data and schema constructs, we use a data structure, Lineage, to denote lineage data in a schema construct. Each Lineage object has six attributes:

1. *data*, which can be a collection storing materialised lineage data, or, if the lineage data is virtual, it will be the value *null* denoting virtual data;
2. *construct*, which is the name of the schema construct containing the lineage data;
3. *isVirtualData*, stating if the lineage data is virtual or not;
4. *isVirtualConstruct*, stating if the construct is virtual or not;
5. *elemStruct*, describing the structure of the data in the extent of the schema construct, *e.g.*, a 2-item tuple `{x1,x2}`, or a 3-item tuple `{x1,x2,x3}`; this will be *null* if the lineage data is materialised.
6. *constraint*, expressing a constraint which derives the lineage data from the schema construct if the construct is virtual; this will be *null* if the lineage data is materialised.

For example, supposing lineage data in a schema construct `D` is derived from the query [`{x,y}|{x,y}` ← `D`; `x=5`], and lp is a Lineage object which expresses this lineage data. If `D=[{1,2},{5,1},{5,2},{3,1}]` is materialised, then lp will be:

lp.*data*	=	`[{5,1},{5,2}]`
lp.*construct*	=	"`D`"
lp.*isVirtualData*	=	$false$
lp.*isVirtualConstruct*	=	$false$
lp.*elemStruct*	=	$null$
lp.*constraint*	=	$null$

On the other hand, If `D` is a virtual schema construct, then lp will be:

lp.*data*	=	*null*
lp.*construct*	=	"`D`"
lp.*isVirtualData*	=	*true*
lp.*isVirtualConstruct*	=	*true*
lp.*elemStruct*	=	"`{x,y}`"
lp.*constraint*	=	"`x=5`"

For ease of exposition, we denote by `O`|`dl` a Lineage object in which `O` is the name of the schema construct and `dl` is the lineage data. If the lineage data is materialised, `dl` will be the data itself. Otherwise `dl` will be the form of (S, C), where S denotes the *elemStruct* and C the *constraint*. For example, the above two Lineage objects are denoted by `D`|`[{5,1},{5,2}]` and `D`|(`{x,y}`,`x=5`), respectively.

In order to express a transformation step with a virtual result or virtual data sources, we use a data structure, TransfStep, to express transformation steps. Each TransfStep object has six attributes:

1. *action*, which can be "*add*", "*del*", "*rename*", "*extend*" and "*contract*";
2. *query*, showing the query used in this transformation step;
3. *result*, which is the name of the schema construct created by this transformation step (if the *action* is "*add*", "*rename*" or "*extend*"), or the name of the construct deleted by this step (if the *action* is "*del*" or "*contract*");
4. *vResult*, stating if the result construct is virtual or not;
5. *sources*, containing all schema construct schemes appearing in the query;
6. *vSources*, a Boolean array, showing which source constructs in the *sources* collection are virtual.

For example, supposing ts is a TransfStep object, where

$$
\begin{aligned}
ts.\mathsf{action} &= \text{"}add\text{"}\\
ts.\mathsf{query} &= \text{"}\langle\langle \mathsf{staff}, \mathsf{name}\rangle\rangle ++ \langle\langle \mathsf{student}, \mathsf{name}\rangle\rangle ++ \langle\langle \mathsf{visitor}, \mathsf{name}\rangle\rangle\text{"}\\
ts.\mathsf{result} &= \text{"}\langle\langle \mathsf{faculty}, \mathsf{name}\rangle\rangle\text{"}\\
ts.\mathsf{vResult} &= true\\
ts.\mathsf{sources} &= [\langle\langle \mathsf{staff}, \mathsf{name}\rangle\rangle, \langle\langle \mathsf{student}, \mathsf{name}\rangle\rangle, \langle\langle \mathsf{visitor}, \mathsf{name}\rangle\rangle]\\
ts.\mathsf{vSources} &= [false, true, false]
\end{aligned}
$$

This means ts is an add transformation creating a new virtual construct ⟨⟨faculty, name⟩⟩ defined by the query "⟨⟨staff, name⟩⟩++⟨⟨student, name⟩⟩++⟨⟨visitor, name⟩⟩". The data sources of ts are ⟨⟨staff, name⟩⟩, ⟨⟨student, name⟩⟩ and ⟨⟨visitor, name⟩⟩, in which ⟨⟨student, name⟩⟩ is virtual and the other two are materialised.

6.3 DLT for a Single Transformation Step

We now investigate how to obtain the lineage of the tracing data along a single transformation step which may involve virtual data sources. We only consider add transformations here and extend transformations are discussed in Section 6.6.3. We assume all queries appearing in transformation steps are SIQL queries. Figure 6.1 gives our DLT procedure for a single transformation step, DLT4AStep, where either the tracing data or the data sources may be virtual. The output of DLT4AStep(td,ts) is the lineage data of tracing data td in the data sources of transformation step ts, which is a list of Lineage objects that may contain materialised or virtual lineage data.

We see from Figure 6.1 that our DLT formulae need to handle four cases: MtMs — both the tracing data and the source data are materialised; MtVs — the tracing data is materialised and the source data is virtual; VtMs — the tracing data is virtual and the source data is materialised; and VtVs — both the tracing data and the source data are virtual.

```
Proc DLT4AStep($td, ts$)
{
    $lpList = \emptyset$;
    case MtMs:
      $lpList$ ←DTL formulae for MtMs;
    case MtVs:
      $lpList$ ←DTL formulae for MtVs;
    case VtMs:
      if ($ts$.result is required)
        $mv \leftarrow$ evaluate($ts$.query);     /*recovering $ts$.result
        $td$.data $\leftarrow mv|td$.data;        /*recovering $td$
        $lpList$ ←DTL formulae for MtMs;
      else
        $lpList$ ←DTL formulae for VtMs;
    case VtVs:
      if ($td$ must be materialised)
        $mv \leftarrow$ GQP($ts$.result);         /*recovering $ts$.result
        $td$.data $\leftarrow mv|td$.data;        /*recovering $td$
        $lpList$ ←DTL formulae for MtVs;
      else
        $lpList$ ←DTL formulae for VtVs;
    return $lpList$;
}
```

Figure 6.1: The DLT4AStep Algorithm

In some cases lineage data are untraceable if the tracing data is virtual (see Section 6.4 below for details). In such cases, expressed as conditions "(ts.result is required)" and "(td must be materialised)" in Figure 6.1, we have to recover the tracing data by materialising the result of the transformation step. In the case of VtMs, we use the procedure evaluate to evaluate the query of the transformation step since all data sources are available, while in the case of VtVs, we use AutoMed's global query processor, GQP, to compute the result from the original data sources.

6.4 DLT Formulae

This section gives our DLT formulae for tracing data lineage for the four cases discussed above: MtMs, MtVs, VtMs and VtVs. The DLT formulae for the case of MtMs are given in Table 6.1 which is a summary of the DLT formulae described in Chapter 5. The DLT formulae in Table 6.1 either provide a derivation tracing query specifying the lineage data of a tracing tuple t or, in some cases, give the lineage data itself directly. If the DLT formula returns a derivation tracing query, we need to evaluate the query to obtain the lineage data. If the formula returns the lineage data directly, no such evaluation is needed.

Since the results of queries of the form `group D` and `gc f D` are a collection of pairs, in the DLT formulae for these two queries we assume that the tracing tuple t is of the form $\{\overline{a}, \overline{b}\}$, where $\overline{a}$ and $\overline{b}$ are patterns. In the last but one line, the notation $\texttt{D}_2|\emptyset$ denotes no lineage in the data source $\texttt{D}_2$.

v	$DL^{AP}(t)$	$DL^{OP}(t)$
`group D`	$[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}; x = \overline{a}]$	
`sort/distinct D`	$\texttt{D}\|t$	
`max/min D`	$\texttt{D}$	$\texttt{D}\|t$
`sum D`	$\texttt{D}$	$[x \mid x \leftarrow \texttt{D}; x \neq 0]$
`count/avg D`	$\texttt{D}$	
`gc max/min D`	$[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}; x = \overline{a}]$	$\texttt{D}\|t$
`gc sum D`	$[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}; x = \overline{a}]$	$[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}; x = \overline{a}; y \neq 0]$
`gc count/avg D`	$[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}; x = \overline{a}]$	
$\texttt{D}_1 \mathbin{++} \texttt{D}_2 \mathbin{++} \ldots \mathbin{++} \texttt{D}_n$	$\forall i.\texttt{D}_i\|t$	
$\texttt{D}_1 \mathbin{--} \texttt{D}_2$	$\texttt{D}_1\|t,\ \texttt{D}_2$	$\texttt{D}_1\|t,\ \texttt{D}_2\|t$
$[\overline{\texttt{x}} \mid \overline{\texttt{x}_1} \leftarrow \texttt{D}_1; \ldots; \overline{\texttt{x}_n} \leftarrow \texttt{D}_n; \texttt{C}]$	$\forall i.[x_i \mid x_i \leftarrow \texttt{D}_i; x_i = ((\texttt{lambda}\ \overline{x}.\overline{x_i})\ t)]$	
$[\overline{\texttt{x}} \mid \overline{\texttt{x}} \leftarrow \texttt{D}_1;\ \texttt{member}\ \texttt{D}_2\ \overline{\texttt{y}}]$	$\texttt{D}_1\|t,\ [y \mid y \leftarrow \texttt{D}_2; y = ((\texttt{lambda}\ \overline{x}.\overline{y})\ t)]$	
$[\overline{\texttt{x}} \mid \overline{\texttt{x}} \leftarrow \texttt{D}_1;\ \texttt{not(member}\ \texttt{D}_2\ \overline{\texttt{y}})]$	$\texttt{D}_1\|t,\ \texttt{D}_2$	$\texttt{D}_1\|t,\ \texttt{D}_2\|\emptyset$
$\texttt{map (lambda p}_1.\texttt{p}_2\texttt{) D}$	$[p_1 \mid p_1 \leftarrow \texttt{D}, p_2 = t]$	

Table 6.1: DLT Formulae for MtMs

From the formulae for MtMs we have derived the DLT formulae for the other three cases below.

6.4.1 Case MtVs

Recall that there two kinds of DLT formulae in Table 6.1: tracing queries and real lineage data. With MtVs the source data is virtual, so we cannot evaluate tracing queries and Lineage objects are required to store the information about these queries. For example, the tracing query [`{x,y}|{x,y}` $\leftarrow$ `D;x=ā`] is expressed as `D|({x,y},x=ā)`, and the corresponding Lineage object, lp, is

lp.*data*	=	*null*
lp.*construct*	=	"`D`"
lp.*isVirtualData*	=	*true*
lp.*isVirtualConstruct*	=	*true*
lp.*elemStruct*	=	"`{x,y}`"
lp.*constraint*	=	"`x=ā`"

In the case of real lineage data, the lineage data may be the tracing data, t, itself or all the items in a source collection `D`. If the lineage data is t, it is available no matter whether `D` is materialised or not. If the lineage data is all items in a virtual collection `D`, it is expressed by `D|(any,true)`, and the corresponding Lineage object, lp, is:

lp.*data*	=	*null*
lp.*construct*	=	"`D`"
lp.*isVirtualData*	=	*true*
lp.*isVirtualConstruct*	=	*true*
lp.*elemStruct*	=	*null*
lp.*constraint*	=	*null*

Table 6.2 gives the DLT formulae for the case of MtVs.

$\mathtt{v}$	$DL^{AP}(t)$	$DL^{OP}(t)$
$\mathtt{group\ D}$	$\mathtt{D}\vert(\{x,y\}, x=\overline{a})$	
$\mathtt{sort/distinct\ D}$	$\mathtt{D}\vert t$	
$\mathtt{max/min\ D}$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	$\mathtt{D}\vert t$
$\mathtt{sum\ D}$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	$\mathtt{D}\vert(x, x \neq 0)$
$\mathtt{count/avg\ D}$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	
$\mathtt{gc\ max/min\ \ D}$	$\mathtt{D}\vert(\{x,y\}, x=\overline{a})$	$\mathtt{D}\vert t$
$\mathtt{gc\ sum\ \ D}$	$\mathtt{D}\vert(\{x,y\}, x=\overline{a})$	$\mathtt{D}\vert(\{x,y\}, x=\overline{a}, y \neq 0)$
$\mathtt{gc\ count/avg\ \ D}$	$\mathtt{D}\vert(\{x,y\}, x=\overline{a})$	
$\mathtt{D}_1 \mathbin{++} \mathtt{D}_2 \mathbin{++} \ldots \mathbin{++} \mathtt{D}_n$	$\forall i.\mathtt{D}_i\vert t$	
$\mathtt{D}_1 -- \mathtt{D}_2$	$\mathtt{D}_1\vert t$, $\mathtt{D}_2\vert(\mathtt{any},\mathtt{true})$	$\mathtt{D}_1\vert t$, $\mathtt{D}_2\vert t$
$[\overline{\mathtt{x}}\vert\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1;\ldots;\overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n;\mathtt{C}]$	$\forall i.\mathtt{D}_i\vert(x_i, x_i = ((\mathtt{lambda}\ \overline{x}.\overline{x_i})\ t))$	
$[\overline{\mathtt{x}}\vert\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;\ \mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}}]$	$\mathtt{D}_1\vert t$, $\mathtt{D}_2\vert(y, y = ((\mathtt{lambda}\ \overline{x}.\overline{y})\ t))$	
$[\overline{\mathtt{x}}\vert\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;\ \mathtt{not(member}\ \mathtt{D}_2\ \overline{\mathtt{y}})]$	$\mathtt{D}_1\vert t$, $\mathtt{D}_2\vert(\mathtt{any},\mathtt{true})$	$\mathtt{D}_1\vert t$, $\mathtt{D}_2\vert\emptyset$
$\mathtt{map\ (lambda\ p_1.p_2)\ D}$	$\mathtt{D}\vert(p_1, p_2 = t)$	

Table 6.2: DLT Formulae for MtVs

We can see that, in Table 6.2, although data sources are virtual, the lineage data is materialised, and so not all computed lineage data is virtual. For example, the affect-pool for aggregate functions are all the tuples in the source collection, *i.e.* `D|(any,true)` (virtual lineage data); the affect-pool for `group` and `gc aggFun` are all the tuples in the source collection whose first component is $\overline{\mathtt{a}}$, *i.e.* `D|({x,y},x=`$\overline{\mathtt{a}}$`)` (again virtual lineage data); while the affect-pool for `sort`, `distinct` and ++ is the tracing data itself, *i.e.* $\mathtt{D}\vert t$ (materialised lineage data).

We note that, in the case of $\mathtt{D}_1\mathbin{++}\mathtt{D}_2\mathbin{++}\ldots\mathbin{++}\mathtt{D}_n$, if a data source $\mathtt{D}_i$ is virtual, we need to compute $\mathtt{D}_i$ to determine if it contains the tracing data t or not. We may materialise all data sources of ++ queries, so as to change the case into MtMs and solve the problem. However, in some cases, tracing data lineage of ++ queries is possible with virtual data sources. For example, suppose $\mathtt{v} = \mathtt{v1} \mathbin{++} \mathtt{D}_1$ and $\mathtt{v1} = \mathtt{distinct}\ \mathtt{D}_2$, in which `v1` is a virtual schema construct and $\mathtt{D}_1$ and $\mathtt{D}_2$ are materialised. In order to trace the lineage of the data in `v`, we actually have no need to materialise `v1`. In particular, we can obtain $\mathtt{v1}\vert t$'s lineage in $\mathtt{D}_2$ as

$\texttt{[x|x} \leftarrow \texttt{D}_2\texttt{;x} = t\texttt{]}$.

In our approach, we retain the data source of ++ as virtual and assume that the lineage data in the virtual data source is t. Then, we use a DLT check process, which is described below, to determine whether the virtual data source needs to be computed[1].

Supposing `S` is a virtual data source of a ++ query, then we firstly find the transformation step, `ts`, that creates `S`. Suppose the query in `ts` is `q`.

If `q` is a ++ query, then the virtual data source `S` can remain virtual, and we have to further check if any of the data sources of `q` are virtual ones.

If `q` is `map`, `sort` or `distinct` with a materialised data source, then `S` can remain virtual. The materialised data source can filter the lineage created in the virtual construct `S` and remove extra lineage data, as shown in the above example.

If `q` is −−, `aggFun`, `group`, `gc group`, comprehension, `member` or `not member`, then `S` must be computed.

Otherwise, if `q` is `map`, `sort` or `distinct` with a virtual data source `S'`, then we cannot determine the situation of `S` based on the current step. We have to find the transformation step `ts`' which creates virtual construct `S'`, and repeat the above check steps to examine the query in `ts`'. If `S'` is able to be virtual, then `S` can also be virtual; if `S'` is not, that means we actually have to compute construct `S`, rather than `S'` itself. Recursively, the final situation of construct `S` can be determined.

The same problem as for ++ may occur for −−. In particular, the situation of tracing the origin-pool in the second argument of the query $\texttt{D}_1 -- \texttt{D}_2$, *i.e.* in $\texttt{D}_2$, is similar to the above and we use the same DLT check process to determine whether $\texttt{D}_2$ can be virtual or not.

[1]The computed data source may or may not be materialised. For the purpose DLT, we use the computed data source once and have no need to materialise it in persistent storage. However, for the purpose of future use, we may materialise it to avoid repeated computations.

6.4.2 Case VtMs

Virtual tracing data can be created by the DLT formulae if data sources are virtual. In particular, there are three kinds of virtual lineage data created in Table 6.2: `(any,true)`, $(\texttt{\{x,y\}},\texttt{x=}\overline{\texttt{a}})$, and $(\texttt{p1},\texttt{p2=t})$. Note that, the lineage data $(\overline{\texttt{x}_i}, \overline{\texttt{x}_i} = ((\texttt{lambda}\ \overline{\texttt{x}}.\overline{\texttt{x}_i})\ \texttt{t}))$ and $(\overline{\texttt{y}}, \overline{\texttt{y}} = ((\texttt{lambda}\ \overline{\texttt{x}}.\overline{\texttt{y}})\ \texttt{t}))$ in the cases of a comprehension (11th line) and a comprehension with `member` filter (12th line) are not virtual. Since `t` is materialised data and tuple $\overline{\texttt{x}}$ contains all variables appearing in $\overline{\texttt{x}_i}$, the expression $(\texttt{lambda}\ \overline{\texttt{x}}.\overline{\texttt{x}_i})\ \texttt{t}$ returns materialised data too.

Tables 6.3, 6.4 and 6.5 illustrate the DLT formulae for VtMs. These can be derived by applying the above three kinds of virtual tracing data, $\texttt{Vt}_1$ = `(any,true)`, $\texttt{Vt}_2 = (\texttt{\{x,y\}},\texttt{x=}\overline{\texttt{a}})$ and $\texttt{Vt}_3 = (\texttt{p1},\texttt{p2=t})$, to the DLT formulae for MtMs given in Table 6.1. In particular, Table 6.3 gives the DLT formulae for tracing the affect-pool and Tables 6.4 and 6.5 give the DLT formulae for tracing the origin-pool. In this case of VtMs, since all source data is materialised, there is no virtual intermediate lineage data created.

For example, suppose `v` is defined by the query `group D`. If the virtual tracing tuple t is $\texttt{Vt}_1$, the affect-pool of t is all data in `D`. If t is $\texttt{Vt}_2$, the affect-pool of t is all tuples in `D` with first component equal to $\overline{\texttt{a}}$. If t is $\texttt{Vt}_3$, the affect-pool of t is all tuples in `D` with first component equal to the first component of the tracing data t. We can see that the virtual view, `v`, is used in this query. Since the source data is materialised, we can easily compute `v` and evaluate the tracing query. However, once the virtual view is computed, the virtual tracing data t can also be materialised. In practice, this situation reverts to the case of MtMs which we discussed earlier.

Although all computed lineage data can be materialised in the case of VtMs, we may leave it as virtual lineage data. For example, if the obtained lineage data is all data in a collection `D`, rather than bring all `D`'s data items into memory to

v	t	$DL^{AP}(t)$
group D	$\texttt{Vt}_1$	D
	$\texttt{Vt}_2$	$[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}; x = a]$
	$\texttt{Vt}_3$	$[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}; \texttt{member}\ [\texttt{first}\ p_1 \mid p_1 \leftarrow \texttt{v}; p_2 = t]\ x]$
sort/dinstinct D	$\texttt{Vt}_1$	D
	$\texttt{Vt}_2$	$[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}; x = a]$
	$\texttt{Vt}_3$	$[p_1 \mid p_1 \leftarrow \texttt{D}; p_2 = t]$
aggFun D	$\texttt{Vt}_1$	D
	$\texttt{Vt}_2$	n/a (t cannot be a tuple)
	$\texttt{Vt}_3$	D
gc aggFun D	$\texttt{Vt}_1$	D
	$\texttt{Vt}_2$	$[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}; x = a]$
	$\texttt{Vt}_3$	$[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}; \texttt{member}\ [\texttt{first}\ p_1 \mid p_1 \leftarrow \texttt{v}; p_2 = t]\ x]$
$\texttt{D}_1 \mathbin{++} \texttt{D}_2 \mathbin{++} \ldots \mathbin{++} \texttt{D}_n$	$\texttt{Vt}_1$	$\forall i.\texttt{D}_i$
	$\texttt{Vt}_2$	$\forall i.[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}_i; x = a]$
	$\texttt{Vt}_3$	$\forall i.[p_1 \mid p_1 \leftarrow \texttt{D}_i; p_2 = t]$
$\texttt{D}_1 -- \texttt{D}_2$	$\texttt{Vt}_1$	$\texttt{D}_1 \mid \texttt{v}$, $\texttt{D}_2$
	$\texttt{Vt}_2$	$\texttt{D}_1 \mid [\{x,y\} \mid \{x,y\} \leftarrow \texttt{v}; x = a]$, $\texttt{D}_2$
	$\texttt{Vt}_3$	$\texttt{D}_1 \mid [p_1 \mid p_1 \leftarrow \texttt{v}; p_2 = t]$, $\texttt{D}_2$
$[\overline{\texttt{x}} \mid \overline{\texttt{x}_1} \leftarrow \texttt{D}_1; \ldots; \overline{\texttt{x}_n} \leftarrow \texttt{D}_n; \texttt{C}]$ $(\texttt{C} \neq \emptyset)$	$\texttt{Vt}_1$	$\forall i.[x_i \mid x_i \leftarrow \texttt{D}_i; \texttt{member}\ (\texttt{map}\ (\texttt{lambda}\ \overline{x}.\overline{x_i})\ \texttt{v})\ x_i]$
	$\texttt{Vt}_2$	$\forall i.[x_i \mid x_i \leftarrow \texttt{D}_i; \texttt{member}\ (\texttt{map}\ (\texttt{lambda}\ \overline{x}.\overline{x_i})$ $[\overline{x} \mid \overline{x} \leftarrow \texttt{v}; \texttt{first}\ \overline{x} = a])\ x_i]$
	$\texttt{Vt}_3$	$\forall i.[x_i \mid x_i \leftarrow \texttt{D}_i;$ $\texttt{member}\ (\texttt{map}\ (\texttt{lambda}\ \overline{x}.\overline{x_i})\ [p_1 \mid p_1 \leftarrow \texttt{v}; p_2 = t])\ x_i]$
$[\overline{\texttt{x}} \mid \overline{\texttt{x}} \leftarrow \texttt{D}_1;$ $\texttt{member}\ \texttt{D}_2\ \overline{\texttt{y}}]$	$\texttt{Vt}_1$	$\texttt{D}_1 \mid \texttt{v}$, $[y \mid y \leftarrow \texttt{D}_2; \texttt{member}\ (\texttt{map}\ (\texttt{lambda}\ \overline{x}.\overline{y})\ \texttt{v})\ y]$
	$\texttt{Vt}_2$	$[\overline{x} \mid \overline{x} \leftarrow \texttt{D}_1; \texttt{member}\ \texttt{D}_2\ \overline{y}; \texttt{first}\ \overline{x} = a]$,$[y \mid y \leftarrow \texttt{D}_2;$ $\texttt{member}\ (\texttt{map}\ (\texttt{lambda}\ x.y)\ [x \mid x \leftarrow \texttt{v}; \texttt{first}\ x = a])\ y]$
	$\texttt{Vt}_3$	$[\overline{x} \mid \overline{x} \leftarrow \texttt{D}_1; \texttt{member}\ \texttt{D}_2\ \overline{y}; e = t]$,$[y \mid y \leftarrow \texttt{D}_2;$ $\texttt{member}\ (\texttt{map}\ (\texttt{lambda}\ \overline{x}.y)\ [p_1 \mid p_1 \leftarrow \texttt{v}; p_2 = t])\ y]$
$[\overline{\texttt{x}} \mid \overline{\texttt{x}} \leftarrow \texttt{D}_1;$ $\texttt{not}(\texttt{member}\ \texttt{D}_2\ \overline{\texttt{y}})]$	$\texttt{Vt}_1$	$\texttt{D}_1 \mid \texttt{v}$, $\texttt{D}_2$
	$\texttt{Vt}_2$	$\texttt{D}_1 \mid [\{x,y\} \mid \{x,y\} \leftarrow \texttt{v}; x = a]$, $\texttt{D}_2$
	$\texttt{Vt}_3$	$\texttt{D}_1 \mid [p_1 \mid p_1 \leftarrow \texttt{v}; p_2 = t]$, $\texttt{D}_2$
$\texttt{map}\ (\texttt{lambda}\ \texttt{p}'_1.\texttt{p}'_2)\ \texttt{D}$	$\texttt{Vt}_1$	D
	$\texttt{Vt}_2$	$[p'_1 \mid p'_1 \leftarrow \texttt{D}; (\texttt{first}\ p'_2) = a]$
	$\texttt{Vt}_3$	$[p'_1 \mid p'_1 \leftarrow \texttt{D}; p_2 = t]$

$\mathsf{Vt}_1 = (\mathsf{any}, \mathsf{true})$, $\mathsf{Vt}_2 = (\{\mathsf{x}, \mathsf{y}\}, \mathsf{x} = \mathsf{a})$, $\mathsf{Vt}_3 = (\mathsf{p}_1, \mathsf{p}_2 = \mathsf{t})$

Table 6.3: DLT Formulae for Tracing the Affect-Pool of VtMs

v	t	$DL^{OP}(t)$
`group D`	$\mathtt{Vt}_1$	$\mathtt{D}$
	$\mathtt{Vt}_2$	$[\{x, y\} \vert \{x, y\} \leftarrow \mathtt{D}; x = a]$
	$\mathtt{Vt}_3$	$[\{x, y\} \vert \{x, y\} \leftarrow \mathtt{D}; \mathtt{member}\ [\mathtt{first}\ p_1 \vert p_1 \leftarrow \mathtt{v}; p_2 = t]\ x]$
`sort/distinct D`	$\mathtt{Vt}_1$	$\mathtt{D}$
	$\mathtt{Vt}_2$	$[\{x, y\} \vert \{x, y\} \leftarrow \mathtt{D}; x = a]$
	$\mathtt{Vt}_3$	$[p_1 \vert p_1 \leftarrow \mathtt{D}; p_2 = t]$
`max/min D`	$\mathtt{Vt}_1$	$\mathtt{D} \vert \mathtt{v}$
	$\mathtt{Vt}_2$	n/a (t cannot be a tuple)
	$\mathtt{Vt}_3$	$\mathtt{D} \vert \mathtt{v}$
`sum D`	$\mathtt{Vt}_1$	$[x \vert x \leftarrow \mathtt{D}; x \neq 0]$
	$\mathtt{Vt}_2$	n/a (t cannot be a tuple)
	$\mathtt{Vt}_3$	$[x \vert x \leftarrow \mathtt{D}; x \neq 0]$
`count/avg D`	$\mathtt{Vt}_1$	$\mathtt{D}$
	$\mathtt{Vt}_2$	n/a (t cannot be a tuple)
	$\mathtt{Vt}_3$	$\mathtt{D}$
`gc max/min  D`	$\mathtt{Vt}_1$	$\mathtt{D} \vert \mathtt{v}$
	$\mathtt{Vt}_2$	$\mathtt{D} \vert [\{x, y\} \vert \{x, y\} \leftarrow \mathtt{v}; x = a]$
	$\mathtt{Vt}_3$	$\mathtt{D} \vert [p_1 \vert p_1 \leftarrow \mathtt{v}; p_2 = t]$
`gc sum  D`	$\mathtt{Vt}_1$	$[\{x, y\} \vert \{x, y\} \leftarrow \mathtt{D}; y \neq 0]$
	$\mathtt{Vt}_2$	$[\{x, y\} \vert \{x, y\} \leftarrow \mathtt{D}; x = a; y \neq 0]$
	$\mathtt{Vt}_3$	$[\{x, y\} \vert \{x, y\} \leftarrow \mathtt{D};$ $\mathtt{member}\ [\mathtt{first}\ p_1 \vert p_1 \leftarrow \mathtt{v}; p_2 = t]\ x; y \neq 0]$
`gc count/avg  D`	$\mathtt{Vt}_1$	$\mathtt{D}$
	$\mathtt{Vt}_2$	$[\{x, y\} \vert \{x, y\} \leftarrow \mathtt{D}; x = a]$
	$\mathtt{Vt}_3$	$[\{x, y\} \vert \{x, y\} \leftarrow \mathtt{D}; \mathtt{member}\ [\mathtt{first}\ p_1 \vert p_1 \leftarrow \mathtt{v}; p_2 = t]\ x]$

$\mathsf{Vt}_1 = (\mathsf{any}, \mathsf{true})$, $\mathsf{Vt}_2 = (\{\mathsf{x}, \mathsf{y}\}, \mathsf{x} = \mathsf{a})$, $\mathsf{Vt}_3 = (\mathsf{p}_1, \mathsf{p}_2 = \mathsf{t})$

Table 6.4: DLT Formulae for Tracing the Origin-Pool of VtMs (1)

$\texttt{D}_1 \mathbin{++} \texttt{D}_2 \mathbin{++} \ldots \mathbin{++} \texttt{D}_n$	$\texttt{Vt}_1$	$\forall i.\texttt{D}_i$
	$\texttt{Vt}_2$	$\forall i.[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}_i; x = a]$
	$\texttt{Vt}_3$	$\forall i.[p_1 \mid p_1 \leftarrow \texttt{D}_i; p_2 = t]$
$\texttt{D}_1 -- \texttt{D}_2$	$\texttt{Vt}_1$	$\texttt{D}_1 \mid \texttt{v}$, $\texttt{D}_2 \mid \texttt{v}$
	$\texttt{Vt}_2$	$\texttt{D}_1 \mid [\{x,y\} \mid \{x,y\} \leftarrow \texttt{v}; x = a]$, $[\{x,y\} \mid \{x,y\} \leftarrow \texttt{D}_2; \texttt{member v } \{x,y\}; x = a]$
	$\texttt{Vt}_3$	$\texttt{D}_1 \mid [p_1 \mid p_1 \leftarrow \texttt{v}; p_2 = t]$, $[p_1 \mid p_1 \leftarrow \texttt{D}_2; \texttt{member v } p_1; p_2 = t]$
$[\overline{\texttt{x}} \mid \overline{\texttt{x}_1} \leftarrow \texttt{D}_1; \ldots; \overline{\texttt{x}_n} \leftarrow \texttt{D}_n; \texttt{C}]$ $(\texttt{C} \neq \emptyset)$	$\texttt{Vt}_1$	$\forall i.[x_i \mid x_i \leftarrow \texttt{D}_i; \texttt{member (map (lambda } \overline{x}.\overline{x_i}\texttt{) v) } x_i]$
	$\texttt{Vt}_2$	$\forall i.[x_i \mid x_i \leftarrow \texttt{D}_i; \texttt{member (map (lambda } \overline{x}.\overline{x_i}\texttt{) } [\overline{x} \mid \overline{x} \leftarrow \texttt{v}; \texttt{first } \overline{x} = a]\texttt{) } x_i]$
	$\texttt{Vt}_3$	$\forall i.[x_i \mid x_i \leftarrow \texttt{D}_i; \texttt{member (map (lambda } \overline{x}.\overline{x_i}\texttt{) } [p_1 \mid p_1 \leftarrow \texttt{v}; p_2 = t]\texttt{) } x_i]$
$[\overline{\texttt{x}} \mid \overline{\texttt{x}} \leftarrow \texttt{D}_1; \texttt{member D}_2\ \overline{\texttt{y}}]$	$\texttt{Vt}_1$	$\texttt{D}_1 \mid \texttt{v}$, $[y \mid y \leftarrow \texttt{D}_2; \texttt{member (map (lambda } \overline{x}.\overline{y}\texttt{) v) } y]$
	$\texttt{Vt}_2$	$[\overline{x} \mid \overline{x} \leftarrow \texttt{D}_1; \texttt{member D}_2\ \overline{y}; \texttt{first } \overline{x} = a]$, $[y \mid y \leftarrow \texttt{D}_2; \texttt{member (map (lambda } x.y\texttt{) } [x \mid x \leftarrow \texttt{v}; \texttt{first } x = a]\texttt{) } y]$
	$\texttt{Vt}_3$	$[p_1 \mid p_1 \leftarrow \texttt{D}_1; \texttt{member D}_2\ \overline{y}; p_2 = t]$,$[y \mid y \leftarrow \texttt{D}_2; \texttt{member (map (lambda } \overline{x}.y\texttt{) } [p_1 \mid p_1 \leftarrow \texttt{v}; p_2 = t]\texttt{) } y]$
$[\overline{\texttt{x}} \mid \overline{\texttt{x}} \leftarrow \texttt{D}_1; \texttt{not(member D}_2\ \overline{\texttt{y}}\texttt{)}]$	$\texttt{Vt}_1$	$\texttt{D}_1 \mid \texttt{v}$, $\texttt{D}_2 \mid \emptyset$
	$\texttt{Vt}_2$	$\texttt{D}_1 \mid [\{x,y\} \mid \{x,y\} \leftarrow \texttt{v}; x = a]$, $\texttt{D}_2 \mid \emptyset$
	$\texttt{Vt}_3$	$\texttt{D}_1 \mid [p_1 \mid p_1 \leftarrow \texttt{v}; p_2 = t]$, $\texttt{D}_2 \mid \emptyset$
$\texttt{map (lambda p}'_1.\texttt{p}'_2\texttt{) D}$	$\texttt{Vt}_1$	$\texttt{D}$
	$\texttt{Vt}_2$	$[p'_1 \mid p'_1 \leftarrow \texttt{D}; (\texttt{first } p'_2) = a)]$
	$\texttt{Vt}_3$	$[p'_1 \mid p'_1 \leftarrow \texttt{D}; p_2 = t]$

$\mathsf{Vt}_1 = (\mathsf{any}, \mathsf{true})$, $\mathsf{Vt}_2 = (\{\mathsf{x}, \mathsf{y}\}, \mathsf{x} = \mathsf{a})$, $\mathsf{Vt}_3 = (\mathsf{p}_1, \mathsf{p}_2 = \mathsf{t})$

Table 6.5: DLT Formulae for Tracing the Origin-Pool of VtMs (2)

continue the DLT process, we may use virtual lineage data, `D|(any,true)`, for the subsequent DLT steps. The materialised lineage data can be extracted from the data sources at the end of the DLT process. This can save the time and memory overheads of the DLT process.

Thus, in practice, we use virtual lineage data even if the data source is materialised and lineage data are materialised only at the end of the DLT process, or, in the case of lineage data that must be materialised in some untraceable cases when the tracing data and data sources are all virtual (the case of VtVs discussed below).

6.4.3 Case **VtVs**

The DLT formulae for VtVs are similar to the formulae for VtMs but in this case the source data are unavailable. Thus, we use Lineage objects to store the virtual intermediate lineage data. However, since data sources are virtual, we cannot compute the virtual view by evaluating the query. Thus, if the virtual view is used in a DLT formula, the lineage data is untraceable without computing the virtual view. Table 6.6 gives the DLT formulae for the case of VtVs.

For example, suppose the query is `v = group D` where `D` is a virtual data source. If the virtual tracing tuple t is `(any,true)`, the virtual affect-pool is `D|(any,true)`. If t is `({x,y},x=`$\overline{a}$`)`, the virtual affect-pool is `D|({x,y},x=`$\overline{a}$`)`. If t is `(p1,p2=t)`, based on the formulae in Table 6.3, the virtual affect-pool in `D` can be expressed as `D|({x,y}, member [first p1|p1` $\leftarrow$ `v;p2=t] x)`. However, we cannot compute `v` by just evaluating the query `group D` defining `v` since `D` is virtual. In this case, AutoMed's Global Query Processor can be used to compute `v`. Once `v` is computed, the virtual tracing data t can also be computed and this situation reverts to the case of MtVs which we discussed earlier.

v	t	$DL^{AP}(t)$	$DL^{OP}(t)$
group D	$\mathtt{Vt}_1$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	
	$\mathtt{Vt}_2$	$\mathtt{D}\vert(\{x,y\},x=a)$	
	$\mathtt{Vt}_3$	untraceable	
sort/distinct D	$\mathtt{Vt}_1$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	
	$\mathtt{Vt}_2$	$\mathtt{D}\vert(\{x,y\},x=a)$	
	$\mathtt{Vt}_3$	$\mathtt{D}\vert(p_1,p_2=t)$	
max/min D	$\mathtt{Vt}_{1,2,3}$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	untraceable
sum D	$\mathtt{Vt}_{1,2,3}$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	$\mathtt{D}\vert(x,x\neq 0)$
count/avg D	$\mathtt{Vt}_{1,2,3}$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	
gc max/min D	$\mathtt{Vt}_1$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	untraceable
	$\mathtt{Vt}_2$	$\mathtt{D}\vert(\{x,y\},x=a)$	untraceable
	$\mathtt{Vt}_3$	untraceable	
gc sum D	$\mathtt{Vt}_1$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	$\mathtt{D}\vert(\{x,y\},y\neq 0)$
	$\mathtt{Vt}_2$	$\mathtt{D}\vert(\{x,y\},x=a)$	$\mathtt{D}\vert(\{x,y\},x=a;y\neq 0)$
	$\mathtt{Vt}_3$	untraceable	
gc count/avg D	$\mathtt{Vt}_1$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	
	$\mathtt{Vt}_2$	$\mathtt{D}\vert(\{x,y\},x=a)$	
	$\mathtt{Vt}_3$	untraceable	
$\mathtt{D}_1 +\!+\, \mathtt{D}_2 +\!+ \ldots +\!+\, \mathtt{D}_n$	$\mathtt{Vt}_1$	$\forall i.\mathtt{D}_i\vert(\mathtt{any},\mathtt{true})$	
	$\mathtt{Vt}_2$	$\forall i.\mathtt{D}_i\vert(\{x,y\},x=a)$	
	$\mathtt{Vt}_3$	$\forall i.\mathtt{D}_i\vert(p_1,p_2=t)$	
$\mathtt{D}_1 -\!-\, \mathtt{D}_2$	$\mathtt{Vt}_{1,2,3}$	untraceable	
$[\overline{\mathtt{x}}\vert\overline{\mathtt{x}_1}\leftarrow\mathtt{D}_1;\ldots;\overline{\mathtt{x}_n}\leftarrow\mathtt{D}_n;\mathtt{C}]$ $(\mathtt{C}\neq\emptyset)$	$\mathtt{Vt}_{1,2,3}$	untraceable	
$[\overline{\mathtt{x}}\vert\overline{\mathtt{x}}\leftarrow\mathtt{D}_1;\ \mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}}]$	$\mathtt{Vt}_{1,2,3}$	untraceable	
$[\overline{\mathtt{x}}\vert\overline{\mathtt{x}}\leftarrow\mathtt{D}_1;\ \mathtt{not}(\mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}})]$	$\mathtt{Vt}_{1,2,3}$	untraceable	
map (lambda $\mathtt{p}'_1.\mathtt{p}'_2$) D	$\mathtt{Vt}_1$	$\mathtt{D}\vert(\mathtt{any},\mathtt{true})$	
	$\mathtt{Vt}_2$	$\mathtt{D}\vert(p'_1,(\mathtt{first}\ p'_2)=a)$	
	$\mathtt{Vt}_3$	$\mathtt{D}\vert(p'_1,p_2=t)$	

$\mathsf{Vt}_1=(\mathsf{any},\mathsf{true})$, $\mathsf{Vt}_2=(\{\mathsf{x},\mathsf{y}\},\mathsf{x}=\mathsf{a})$, $\mathsf{Vt}_3=(\mathsf{p}_1,\mathsf{p}_2=\mathsf{t})$

Table 6.6: DLT Formulae for VtVs

Alternatively, the view definition of v could be propagated through the remaining DLT steps until the end of the process. So far we have only implemented the first approach and it remains to implement the second approach and to investigate their trade-offs.

6.5 DLT for General Transformation Pathways

Having obtained the DLT formulae for the above four cases, lineage data based on a single transformation step is obtained by procedure DLT4AStep(td, ts) as described in Section 6.3, and its output is the lineage of td in ts's data sources *i.e.* a list of Lineage objects which may contain either materialised or virtual lineage data.

In our DLT algorithms for a general transformation pathway, there are two further procedures: tracing the lineage of a single tuple along a transformation pathway and tracing the lineage of a set of tuples along a transformation pathway. This is because the lineage of one Lineage object based on a single transformation step may be a list of Lineage objects, if the step has multiple data sources.

6.5.1 The DLT Algorithms

Figure 6.2 presents our DLT algorithms for tracing data lineage along a general transformation pathway: oneDLT4APath($td, [ts_1, ..., t_n]$) traces the lineage of a single tracing tuple td along a transformation pathway $[ts_1, ..., t_n]$, and listDLT4APath($[td_1, ..., td_m]$, $[ts_1, ..., ts_n]$) traces the lineage of a list of tracing tuples along a transformation pathway.

oneDLT4APath firstly finds the transformation step, ts_i, which creates the schema construct containing td and then calls DLT4AStep to obtain the lineage of td based on this transformation step. DLT4AStep returns a list of Lineage

```
Proc oneDLT4APath($td, [ts_1, ..., ts_n]$)
{
  $lpList = \emptyset$;
  for $i = n$ downto 1, do
    if ($td$.construct $= ts_i$.result)
      $Num = i$;
      $lpList =$ DLT4AStep($td, ts_i$);
      $continue$; //* End the $for$ loop
  $restTP = [ts_1, ..., ts_{Num}]$;
  return listDLT4APath($lpList, restTP$);
}

Proc listDLT4APath($[td_1, ..., td_m], [ts_1, ..., ts_n]$)
{
  $lpList = \emptyset$;
  for $i = 1$ to $m$, do
    $lpList =$ merge($lpList$, oneDLT4APath($td_i, [ts_1, ..., ts_n]$));
  return $lpList$;
}
```

Figure 6.2: DLT Algorithms for a General Transformation Pathway

objects. After that, oneDLT4APath calls the procedure listDLT4APath to further trace the lineage of this list of Lineage objects along the rest of the transformation pathway (*i.e.* the steps prior to ts_i). oneDLT4APath also returns a list of Lineage objects. listDLT4APath itself calls oneDLT4APath for each item td_i in the tracing data list to find the entire lineage of the whole list based on the transformation pathway.

The merge function in the procedure listDLT4APath is used to avoid duplication of lineage data (as in Chapter 5, Section 4.5.2).

The algorithms in Figure 6.2 are correct in the sense that they give the same result as the DLT algorithms given in Section 5.5.2 in Chapter 5. This is because the DLT formulae described in Section 6.4, which are used in the procedure DLT4AStep computing lineage data based on one add transformation, can be derived from the DLT formulae described in Section 5.4, while procedures oneDLT4APath and listDLT4APath obtain the final lineage data by checking all transformation steps along a transformation pathway in reverse.

Similarly to the DLT algorithms described in Chapter 5, the exact complexity of the overall DLT process in this chapter is $O(n \times m)$ where n is the number of add transformations relevant to the tracing data in the transformation pathway and m is the number of different schema constructs in the computed lineage data.

6.5.2 Example

Suppose that construct ⟨⟨CourseSum, Avg⟩⟩ is generated by the following transformation steps:

```
⟨⟨CourseSum, Avg⟩⟩  =  [{x,y,z}|{x,y,z} ← gc avg
                           ([{{k1,k2},x}|{k1,k2,k3,x} ← ⟨⟨Details, mark⟩⟩])]
⟨⟨Details, Mark⟩⟩   =  [{'IS',k1,k2,x}|{k1,k2,x} ← ⟨⟨IStab, Mark⟩⟩]
                       ++ [{'MA',k1,k2,x}|{k1,k2,x} ← ⟨⟨MAtab, Mark⟩⟩]
```

where constructs ⟨⟨CourseSum, Avg⟩⟩, ⟨⟨MAtab, Mark⟩⟩ and ⟨⟨IStab, Mark⟩⟩ are materialised and construct ⟨⟨Details, Mark⟩⟩ is virtual. The transformation pathway generating ⟨⟨CourseSum, Avg⟩⟩ construct consists of the following sequence of view definitions, where the intermediate constructs `v1`, ..., `v4` and ⟨⟨Details, Mark⟩⟩ are virtual:

```
v1                 = [{'IS',k1,k2,x}|{k1,k2,x} ← ⟨⟨IStab, Mark⟩⟩]
v2                 = [{'MA',k1,k2,x}|{k1,k2,x} ← ⟨⟨MAtab, Mark⟩⟩]
⟨⟨Details, Mark⟩⟩  = v1 ++ v2
v3                 = map (lambda {k,k1,k2,x}.{{k,k1},x}) ⟨⟨Details, Mark⟩⟩
v4                 = gc avg v3
⟨⟨CourseSum, Avg⟩⟩ = map (lambda {{x,y},z}.{x,y,z}) v4
```

Suppose td = `{'MA','MAC01',81}` is a tuple in construct ⟨⟨CourseSum, Avg⟩⟩. Traversing the above transformation pathway in reverse, we obtain td's lineage data, *dl*, with respect to each view as follows:

$$\begin{array}{lll}
 & td & = \langle\langle \mathsf{CourseSum}, \mathsf{Avg}\rangle\rangle|\texttt{\{'MA','MAC01',81\}} \\
\overset{\mathsf{MtVs}}{\Longrightarrow} & \texttt{v4|dl} & = \texttt{v4}|\texttt{\{\{'MA','MAC01'\},81\}} \\
\overset{\mathsf{MtVs}}{\Longrightarrow} & \texttt{v3|dl} & = \texttt{v3}|(\texttt{\{x,y\},x=\{'MA','MAC01'\}}) \\
\overset{\mathsf{VtVs}}{\Longrightarrow} & \langle\langle \mathsf{Details}, \mathsf{Mark}\rangle\rangle|\texttt{dl} & = \langle\langle \mathsf{Details}, \mathsf{Mark}\rangle\rangle| \\
 & & \qquad (\texttt{\{k,k1,k2,x\},\{k='MA';k1='MAC01'\}}) \\
\overset{\mathsf{VtVs}}{\Longrightarrow} & \texttt{v2|dl} & = \texttt{v2}|(\texttt{\{k,k1,k2,x\},\{k='MA';k1='MAC01'\}}), \\
 & \texttt{v1|dl} & = \texttt{v1}|(\texttt{\{k,k1,k2,x\},\{k='MA';k1='MAC01'\}}) \\
\overset{\mathsf{VtMs}}{\Longrightarrow} & \langle\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\rangle|\texttt{dl} & = \langle\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\rangle| \\
 & & \qquad (\texttt{\{k1,k2,x\},\{'MA'='MA';k1='MAC01'\}}) \\
 & \langle\langle \mathsf{IStab}, \mathsf{Mark}\rangle\rangle|\texttt{dl} & = \langle\langle \mathsf{IStab}, \mathsf{Mark}\rangle\rangle| \\
 & & \qquad (\texttt{\{k1,k2,x\},\{'IS'='MA';k1='MAC01'\}})
\end{array}$$

In conclusion, we can see that the lineage from ⟨⟨IStab, Mark⟩⟩ is empty and the lineage from ⟨⟨MAtab, Mark⟩⟩ is obtained by evaluating the tracing query [{k1,k2,x}| {k1,k2,x} ← ⟨⟨MAtab, Mark⟩⟩; 'MA'='MA';k1='MAC01'].

6.5.3 Performance of the DLT Algorithms

In this section, we study the performance of our DLT algorithms by comparing their running times with respect to the number of relevant add steps in the transformation pathway, and with respect to the number of schema constructs in the computed lineage data. Experiments were set up based on an extension of the example given in Section 4.2.3, where the source schema SS contains several relations of the form *deptName*($\underline{\text{emp\_id}}$, emp_name, salary), and the target schema GS contains two relations person($\underline{\text{emp\_id}}$, emp_name, salary, dept) and deptSum($\underline{\text{deptName}}$,avgSalary).

In Figure 6.3, the tracing data is in the construct ⟨⟨person, salary⟩⟩ of the global schema GS, and only one construct in the source schema SS is computed in the data lineage. In order to set up transformation pathways containing increasing

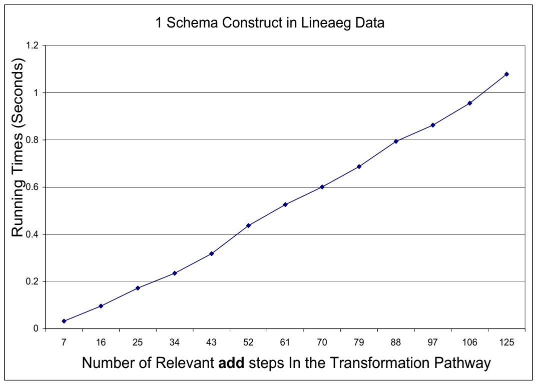


Figure 6.3: Running Time vs. Number of Relevant add Transformations

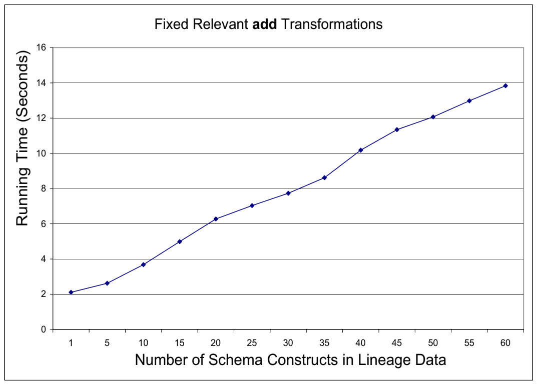


Figure 6.4: Running Time vs. Number of Schema Constructs

numbers of add transformations, we create transformation pathways transforming SS and GS to each other repeatedly, *i.e.* transformation pathways are created in the form of $\texttt{SS} \rightarrow \texttt{GS}_1 \rightarrow \texttt{SS}_1 \rightarrow \texttt{GS}_2 \rightarrow \ldots \rightarrow \texttt{SS}_n \rightarrow \texttt{GS}$, in which $\texttt{SS}_i(i = 1...n)$ is identical to SS and $\texttt{GS}_i(i = 1...n)$ is identical to GS, but only the schemas SS and GS are materialised. Figure 6.3 illustrates the running times of our DLT process based on these transformation pathways[2].

In Figure 6.4, the transformation pathway creating the target schema GS is fixed (and has 16 relevant add transformations). In order to obtain different numbers of constructs in the computed lineage data, we vary the tracing data from containing only one tracing tuple in one global schema construct into a set of tracing tuples from multiple global schema constructs. Figure 6.4 illustrates the running times of our DLT process in this scenario.

We can see that, as expected the running times of our DLT process increase linearly according to the number of relevant add transformations and the number

[2]The implemented algorithm does not include the DLT check process described in Section 6.4.1. We do not expect significant changes of the performance if it is extended to include the DLT check process, since the DLT check process only examines query types of transformation steps, which has a much lower consuming time than DLT processes. However, this still remains to be verified as future work.

of schema constructs in the computed lineage data.

6.6 Extending the DLT Algorithms

In the above algorithms, we only consider IQL^c queries and add and rename transformations. In practice, queries beyond IQL^c and delete, contract and extend transformations may appear in the transformation pathways integrating warehouse data. We now consider how these transformations can also be used for data lineage tracing.

6.6.1 Using Queries beyond IQL^c

Our DLT algorithms handle IQL^c queries in add transformations. Referring back to the Figure 3.5 in Section 3.3 which illustrates the data transformation and integration processes in a typical data warehouse, add transformations for single-source cleansing may contain built-in functions which cannot be handled by our DLT formulae given earlier. In order to go back all the steps to the data source schemas DSS in the staging area, the DLT process may therefore need to handle queries beyond IQL^c.

In particular, suppose the construct c is created by the following transformation step, in which f is a function defined by means of an arbitrary IQL query and $\mathtt{s_1}, ..., \mathtt{s_n}$ are the schemes appearing in the query:

$\mathtt{addT(c, f(s_1, ..., s_n))}$;

There are three cases for tracing the lineage of a tracing tuple $t \in \mathtt{c}$:

1. f is an IQL^c query, in which case the DLT formulae described in this chapter can be used to obtain t's lineage;

2. $\mathtt{n} = \mathtt{1}$ and $\mathtt{f}$ is of the form $\mathtt{f(s_1)} = [\mathtt{h\ x}|\mathtt{x} \leftarrow \mathtt{s_1}; \mathtt{C}]$ for some $\mathtt{h}$ and $\mathtt{C}$, in which case the lineage of t in $\mathtt{s_1}$ is given by:

   $[\mathtt{x}\,|\,\mathtt{x} \leftarrow \mathtt{s_1}; \mathtt{C}; (\mathtt{h\ x}) = t]$

3. For all other cases, we assume that the data lineage of t in the data source $\mathtt{s_i}$ is all data in $\mathtt{s_i}$, for all $1 \leq \mathtt{i} \leq \mathtt{n}$.

6.6.2 Using delete Transformations

The query in a delete transformation specifies how the extent of the deleted construct can be computed from the remaining schema constructs.

delete transformations are useful for DLT when the construct is unavailable. In particular, if a virtual intermediate construct with virtual data sources must be computed during the DLT process, normally we have to use the AutoMed Global Query Processor to derive this construct from the original data sources. However, if the virtual intermediate construct is deleted by a delete transformation and all constructs appearing in the delete transformation are materialised, then we can use the query in the delete transformation to compute the virtual construct. Since we only need to access materialised constructs in the data warehouse, the time of the evaluation procedure is reduced.

This feature can make a view *self-traceable*. That is, for the data in an integrated view, we can identify the names of the source constructs containing the lineage data, and obtain the lineage data from the view itself, rather than access the source constructs.

6.6.3 Using extend Transformations

An extend transformation is applied if the extent of a new construct cannot be precisely derived from the source schema. The transformation $\mathsf{extendT}(\mathsf{c}, ql, qu)$

adds a new construct c to a schema, where query ql determines from the schema what is the minimum extent of c (and may be Void) and qu determines what is the maximal extent of c (and may be Any) [MP03b].

If the transformation is $\mathsf{extendT}(\mathsf{c}, \mathsf{Void}, \mathsf{Any})$, this means that the extent of c is not derived from the source schema. We simply terminate the DLT process for tracing the lineage of c's data at that step.

If the transformation is $\mathsf{extendT}(\mathsf{c}, ql, \mathsf{Any})$, this means the extent of c can be partially computed by the query ql. Using ql, we can obtain a part of the lineage of c's data.

However, we cannot simply treat the DLT process via such an extend transformation as the same as via an add transformation by using the DLT formulae described in Section 6.4. Since in an add transformation, the whole extent of the added construct is exactly specified, while in an extend transformation it is not. The problem that arises is that extra lineage data may be derived because the tracing data contains more data than the result of the query, ql, in the extend transformation.

For example, transformation $\mathsf{extendT}(\mathsf{c}, \mathtt{D}_1 -- \mathtt{D}_2, \mathsf{Any})$, where $\mathtt{D}_1 = [1, 2, 3]$, $\mathtt{D}_2 = [2, 3, 4]$. Although the query result is list [1], the extent of c may be $[1, 2]$, in which $''2''$ is derived from other transformation pathways. If we directly use the DLT algorithm described above, the obtained lineage data of $2 \in \mathsf{c}$ are $\mathtt{D}_1|[2]$ and $\mathtt{D}_2|[2, 3, 4]$. While in fact, the data $''2''$ has no data lineage along this extend transformation.

Therefore, in practice, in order to trace data lineage along an extend transformation with the lower-bound query, ql, the result of the query must be recomputed and be used to filter the tracing data during the DLT process.

If the transformation is $\mathsf{extendT}(\mathsf{c}, \mathsf{Void}, qu)$, this means that the extent of c must be fully computed in the result of the query qu. Although extra data may appear in qu's result, it cannot appear in the extent of c. We use the same approach as described for add transformations to trace lineage of c's data based on qu. However, we have to indicate that, extra lineage data may be created.

Finally, if the transformation is $\mathsf{extendT}(\mathsf{c}, ql, qu)$, we firstly obtain the lineage of c's data based on these two queries, and then return their intersection as the final lineage data, which would be much more accurate but still may not be the exact lineage data.

6.6.4 Using contract Transformations

A $\mathsf{contract}$ transformation removes a construct whose extent cannot be precisely computed by the remaining constructs in the schema. The transformation $\mathsf{contractT}(\mathsf{c}, ql, qu)$ removes a construct c from a schema, where ql determines what is the minimum extent of c, and qu determines what is the maximal extent of c. As with extend, ql may be Void and qu may be Any.

If the transformation is $\mathsf{contractT}(\mathsf{c}, \mathsf{Void}, \mathsf{Any})$, we simply ignore the $\mathsf{contract}$ transformation in our DLT process.

Otherwise, we use the $\mathsf{contract}$ transformation similarly to the way we use delete transformations described above. However, we also have to indicate that if using ql, only partial lineage data can be obtained; if using qu, extra lineage data may be obtained; and if using the intersection of the results of both ql and qu, we can also only obtain an approximate lineage data.

6.7 Implementation

This section describes a set of data warehousing packages for the AutoMed toolkit, which implement the generalised DLT algorithm described in this chapter. These packages use java and the AutoMed Repository API.

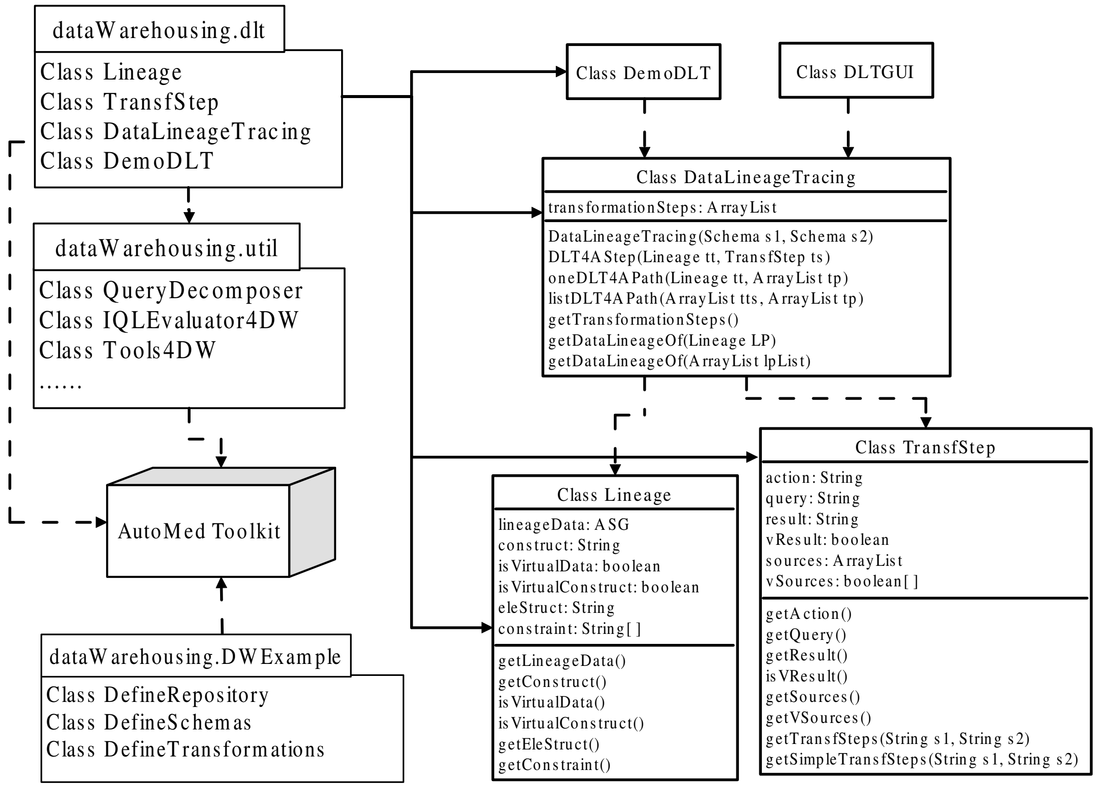


Figure 6.5: The Diagram of the Data Warehousing Toolkit

Currently, there are three packages available in the data warehousing toolkit: dataWarehousing.dlt, dataWarehousing.util and dataWarehousing.DWExample. All packages have the prefixed hierarchy "uk.ac.bbk.automed". The diagram in Figure 6.5 shows the relationships of the three packages and the rest of the AutoMed toolkit, as well as the relationships of the classes in the dataWarehousing.dlt package. Solid arrowed lines indicate the classes contained in the dataWarehousing.dlt package, and dashed arrowed lines indicate the dependence relationships between classes or packages. dataWarehousing.DWExample gives an example of creating the AutoMed metadata for a data warehouse, *i.e.* creating the schemas of the data warehouse and AutoMed transformation pathways expressing mappings between the schemas. dataWarehousing.util includes the utilities used in the data warehousing toolkit. dataWarehousing.dlt contains the class Lineage, which is the data structure storing lineage data; the class TransfStep, which is the data structure storing transformation steps; the class DataLineageTracing, which is the implementation of the generalised DLT algorithm descried in this chapter; and the class DemoDLT, giving an example of using the DLT package. Appendix *C* gives greater details of this data warehousing toolkit.

6.8 Discussion

AutoMed schema transformation pathways can be used to express data transformation and integration processes in heterogeneous data warehousing environments. This chapter has discussed techniques for tracing data lineage along such pathways and thus addresses the general DLT problem for heterogeneous data warehouses.

We have developed a set of DLT formulae using virtual arguments to handle virtual intermediate schema constructs and virtual lineage data. Based on these formulae, our algorithms perform data lineage tracing along a general schema transformation pathway, in which each add transformation step may create either a virtual or a materialised schema construct. In practice, we use virtual data for expressing intermediate lineage data even it is available. This can save the time and memory costs of the DLT processes.

One of the advantages of AutoMed is that its schema transformation pathways can be readily evolved as the data warehouse evolves. In this section we have shown how to perform data lineage tracing along such evolvable pathways.

Furthermore, the Lineage data structure described in Section 6.2 can be used to express the data in the extent of a virtual global schema construct. This extends our DLT method to a virtual data integration framework, where the integrated database is virtual.

Although this chapter has used IQL^c as the query language in which transformations are specified, our algorithms are not limited to one specific data model or query language, and could be applied to other query languages involving common algebraic operations on collections such as selection, projection, join, aggregation, union and difference.

Finally, since our algorithms consider in turn each transformation step in a transformation pathway in order to evaluate lineage data in a stepwise fashion, they are useful not only in data warehousing environments, but also in any data transformation and integration framework based on sequences of primitive schema transformations. For example, [Zam04, ZP04] present an approach for integrating heterogeneous XML documents using the AutoMed toolkit. A schema is automatically extracted for each XML document and transformation pathways are applied to these schemas. Reference [MP03b] also discusses how AutoMed can be applied in peer-to-peer data integration settings. Thus, the DLT approach we have discussed in this chapter is readily applicable in peer-to-peer and semi-structured data integration environments.

Chapter 7

Using AutoMed Transformation Pathways for Incremental View Maintenance

Data warehouses integrate information from distributed, autonomous, and possibly heterogeneous data sources. When data sources are updated, the data warehouse, and in particular the materialised views in the data warehouse, must be updated also. This is the problem of *view maintenance* in data warehouses.

Materialised warehouse views need to be maintained either when the data of a data source changes, or if there is an evolution of a data source schema. Chapter 4 discussed how AutoMed schema transformations can be used to express the evolution of a data source or data warehouse schema, either within the same data model, or a change in its data model, or both; and how the existing warehouse metadata and data can be evolved so that the previous transformation, integration and data materialisation effort can be reused.

In this chapter, we focus on refreshing materialised warehouse views when the data of a data source changes, and we present an incremental view maintenance

(IVM) approach based on AutoMed schema transformation pathways. Section 7.1 discusses related work on view maintenance. Section 7.2 presents our IVM formulae and algorithms over AutoMed schema transformation pathways. Section 7.3 discusses methods for avoiding materialisations in our IVM algorithms. Section 7.4 discusses how queries beyond IQL^c and extend transformations can be used in our IVM process. Finally, Section 7.5 gives our concluding remarks.

7.1 Related Work

The problem of view maintenance at the data level (*i.e.* when the database schema does not change) has been widely discussed in the literature. Comprehensive surveys of this problem are given in [GM99, Don99], as well as a discussion of applications, problems and techniques for maintaining materialised views.

The work of Blakeley *et al.* in [BLT86, BCL89] presents the notion of *irrelevant update* denoting updates applied to source relations that have no effect on the state of the derived relations. They discuss a mechanism of detecting irrelevant updates. As to relevant updates, *i.e.* updates over source relations that may have an effect on the state of the derived relations, an approach for maintaining select-project-join (SPJ) views is presented.

Reference [QW91] presents a set of propagation rules for deriving incremental expressions which compute the changes to SPJ views based on algebraic operations. This work also indicates that these derived incremental expressions are not always cheaper to evaluate than recomputing the views from scratch.

Ceri and Widom's work in [CW91] presents an approach for deriving production rules for maintaining SQL views, but does not consider duplicate data items, aggregate functions, and difference operations. This algorithm determines the key of the source relation that is updated in order to efficiently maintain the views, but cannot be applied if a view does not contain the key attributes from the source relation.

Gupta *et al.*'s work [GMS93] presents a deferred view maintenance algorithm, *counting*, applying to SQL views which may or may not have duplicate data items and can be defined by aggregate functions, and `UNION` and difference operators. This algorithm works by storing the number of the derivations of each tuple in the materialised view.

References [GL95, CGL$^+$96, Qua96] present propagation formulae based on relational algebra operations for incrementally maintaining views with duplicates and aggregations. In particular, reference [CGL$^+$96] describes propagation formulae based on post-update source tables, that is source tables available in the state where changes have already been applied.

Reference [PSCP02] discusses the problem of incrementally maintaining views of *non-distributive* aggregate functions. An aggregate function is *distributive* if the refreshed view can be computed by only using the original view and the changes to the source tables, such as SUM and COUNT. In order to maintain non-distributive aggregate function views, such as AVG, MAX and MIN views after a DELETE operation, not only the changes to the source table, but also the source table itself has to be used in the maintenance process.

The problem of view maintenance in data warehousing environments has been discussed by Zhuge *et al.* in [ZGMHW95, ZGMW96, ZGMW98]. In particular, reference [ZGMHW95] considers the IVM problem for a single-source data warehouse and references [ZGMW96, ZGMW98] for a multi-source data warehouse. Four consistency levels of warehouse data are considered in these works: *convergence* — after the last update and all activity has ceased, the view is consistent with the source relations; *weak consistency* — every state of the view corresponds to some valid state of the source relations, but possibly not in a corresponding order: for example, supposing that the state i and j of the view corresponds to the state p and q of the source relations, it may be that $i < j$ but $p > q$; *strong consistency* — every state of the view corresponds to a valid state of the source relations, and in a corresponding order; and *completeness* — there is a 1-1 order-preserving mapping between the sates of the view and the states of the data sources.

The problem of IVM for multi-source data warehouses has also been discussed in other literature. For example, reference [MS01] presents change propagation rules for IVM of multi-source views which can involve one or more base relations belonging to one or more data sources. Reference [AASY97] presents two IVM algorithms, namely the SWEEP and Nested SWEEP algorithms, focusing on views defined by SPJ expressions. Based on the two SWEEP algorithms, reference [DZR99] develops the MRE Wrapper for incrementally maintaining warehouse views.

In addition, reference [QW97] presents a concurrency control algorithm, *2VNL*, for maintaining on-line data warehouses and allowing user queries and warehouse maintenance transactions to execute concurrently without blocking each other. References [GGMS97, AFP03] discuss the view maintenance problem in the context of object-oriented database systems, where views can be defined by object query languages such as OQL. In particular, reference [AFP03] describes an approach to immediate IVM for OQL views by storing object IDs of source objects.

7.2 IVM over AutoMed Schema Transformations

Our IVM algorithms use the individual steps of a transformation pathway to compute the changes to each intermediate construct in the pathway, and finally obtain the changes to the view created by the transformation pathway in a step-wise fashion. Since no construct in a global schema is contributed by delete and contract transformations, we ignore these transformations in our IVM algorithms. In addition, computing changes based on a transformation $\mathsf{renameT}(\mathsf{O}, \mathsf{O}')$ is simple — the changes to O' are the same as the changes to O. Thus, we only consider add transformations here. In Section 7.4.2 we discuss using also extend transformations.

We develop a set of IVM formulae for each kind of SIQL query that may appear in an add transformation. These IVM formulae can be applied on each add transformation step in order to compute the changes to the construct created by that step. By following all the steps in the transformation pathway, we thus compute the intermediate changes step by step, finally ending up with the final changes to the global schema data.

Referring back to Figure 3.5 in Section 3.3 which illustrates the data transformation and integration processes in a typical data warehouse, in this chapter we assume that the data source updates input to our IVM process are with respect to the single-cleansed schemas SS_i. Thus, our IVM process can be used to maintain those materialised schemas which are downstream from the single-source data cleansing, including the multi-cleansed schemas, data warehouse schemas and data mart schemas.

7.2.1 IVM Formulae for SIQL Queries

We use $\triangle\mathcal{C}/\triangledown\mathcal{C}$ to denote a collection of data items inserted into/deleted from a collection $\mathcal{C}$[1]. There may be many possible expressions for $\triangle\mathcal{C}$ and $\triangledown\mathcal{C}$ but not all are equally desirable. For example, we could simply let $\triangledown\mathcal{C} = \mathcal{C}$ and $\triangle\mathcal{C} = \triangle\mathcal{C}^{new}$, but this is equivalent to recomputing the view from scratch [Qua96]. In order

[1]For the purposes of this chapter, all collections are assumed to be bags.

to guard against such definitions, we use the concept of *minimality* [GL95] to ensure that no unnecessary data are produced.

Minimality Conditions Any changes ($\triangle\mathcal{C}/\triangledown\mathcal{C}$) to a data collection $\mathcal{C}$, including the data source and the view, must satisfy the following minimality conditions:

(i) $\triangledown\mathcal{C} \subseteq \mathcal{C}$: We only delete tuples that are in $\mathcal{C}$;

(ii) $\triangle\mathcal{C} \cap \triangledown\mathcal{C} = \emptyset$: We do not delete a tuple and then reinsert it.

We now give the IVM formulae for each kind of SIQL query, in which `v` denotes the view, `D` denotes the updated data source, $\triangle$`v`/$\triangledown$`v` and $\triangle$`D`/$\triangledown$`D` denote the collections inserted into/deleted from `v` and `D`, and $\mathtt{D}^{new}$ denotes the data source after the update. We observe that these formulae guarantee that the above minimality conditions are satisfied by $\triangle$`v` and $\triangledown$`v` provided they are satisfied by $\triangle$`D` and $\triangledown$`D`.

IVM formulae for distinct, map, and aggregate functions

Table 7.1 illustrates the IVM formulae for these functions. We can see that the IVM formulae for distinct/max/min/avg require access to the post-update data source and using the view data; the formulae for count/sum need to use the view data; and the formulae for map use only the updates to the data source.

IVM formulae for grouping functions

Grouping functions, such as `group D` and `gc f D`, group a bag of pairs `D` on their first component, and may apply an aggregate function `f` to the second component. In order to incrementally maintain a view defined by a grouping function, we firstly find the data items in `D` which are in the same groups as the updates, *i.e.* have the same first component as one or more of the updates. Then

v		IVM Formulae
`distinct D`	$\triangle$v	$\texttt{distinct}\ [x \mid x \leftarrow \triangle\texttt{D};\ \texttt{not}\ (\texttt{member v}\ x)]$
	$\triangledown$v	$\texttt{distinct}\ [x \mid x \leftarrow \triangledown\texttt{D};\ \texttt{not}\ (\texttt{member D}^{new}\ x)]$
`map (lambda p1.p2) D`	$\triangle$v	$\texttt{map}\ (\texttt{lambda p1.p2})\ \triangle\texttt{D}$
	$\triangledown$v	$\texttt{map}\ (\texttt{lambda p1.p2})\ \triangledown\texttt{D}$
`max D`	$\triangle$v	let $\texttt{r1 = max}\ \triangle\texttt{D};\ \texttt{r2 = max}\ \triangledown\texttt{D}$ $\begin{cases} \texttt{max}\ \triangle\texttt{D}, & \text{if } (\texttt{v} < \texttt{r1}); \\ \emptyset, & \text{if } (\texttt{v} \geq \texttt{r1}) \& (\texttt{v} \neq \texttt{r2}); \\ \texttt{max D}^{new}, & \text{if } (\texttt{v} > \texttt{r1}) \& (\texttt{v} = \texttt{r2}). \end{cases}$
	$\triangledown$v	$\begin{cases} \texttt{v}, & \text{if } (\texttt{v} < \texttt{r1}); \\ \emptyset, & \text{if } (\texttt{v} \geq \texttt{r1}) \& (\texttt{v} \neq \texttt{r2}); \\ \texttt{v}, & \text{if } (\texttt{v} > \texttt{r1}) \& (\texttt{v} = \texttt{r2}). \end{cases}$
`min D`	$\triangle$v	let $\texttt{r1 = min}\ \triangle\texttt{D};\ \texttt{r2 = min}\ \triangledown\texttt{D}$ $\begin{cases} \texttt{min}\ \triangle\texttt{D}, & \text{if } (\texttt{v} > \texttt{r1}); \\ \emptyset, & \text{if } (\texttt{v} \leq \texttt{r1}) \& (\texttt{v} \neq \texttt{r2}); \\ \texttt{min D}^{new}, & \text{if } (\texttt{v} < \texttt{r1}) \& (\texttt{v} = \texttt{r2}). \end{cases}$
	$\triangledown$v	$\begin{cases} \texttt{v}, & \text{if } (\texttt{v} > \texttt{r1}); \\ \emptyset, & \text{if } (\texttt{v} \leq \texttt{r1}) \& (\texttt{v} \neq \texttt{r2}); \\ \texttt{v}, & \text{if } (\texttt{v} < \texttt{r1}) \& (\texttt{v} = \texttt{r2}). \end{cases}$
`count D`	$\triangle$v	$\texttt{v} + (\texttt{count}\ \triangle\texttt{D}) - (\texttt{count}\ \triangledown\texttt{D})$
	$\triangledown$v	v
`sum D`	$\triangle$v	$\texttt{v} + (\texttt{sum}\ \triangle\texttt{D}) - (\texttt{sum}\ \triangledown\texttt{D})$
	$\triangledown$v	v
`avg D`	$\triangle$v	$\texttt{avg D}^{new}$
	$\triangledown$v	v

Table 7.1: IVM Formulae for distinct, map, and Aggregate Functions

this smaller data collection can be used to compute the changes to the view, so as to save time and space. Table 7.2 illustrates the IVM formulae for grouping functions.

v	IVM Formulae	
group D	$\triangle$v	group $[\{x,y\}\|\{x,y\} \leftarrow \texttt{D}^{new};$ member $[p\|\{p,q\} \leftarrow (\triangle\texttt{D} \mathbin{++} \triangledown\texttt{D})]\ x]$
	$\triangledown$v	$[\{x,y\}\|\{x,y\} \leftarrow \texttt{v};$ member $[p\|\{p,q\} \leftarrow (\triangle\texttt{D} \mathbin{++} \triangledown\texttt{D})]\ x]$
gc f D	$\triangle$v	gc f $[\{x,y\}\|\{x,y\} \leftarrow \texttt{D}^{new};$ member $[p\|\{p,q\} \leftarrow (\triangle\texttt{D} \mathbin{++} \triangledown\texttt{D})]\ x]$
	$\triangledown$v	$[\{x,y\}\|\{x,y\} \leftarrow \texttt{v};$ member $[p\|\{p,q\} \leftarrow (\triangle\texttt{D} \mathbin{++} \triangledown\texttt{D})]\ x]$

Table 7.2: IVM Formulae for Grouping Functions

We can see that the IVM formulae for grouping functions require access to the updated data source and using the view data.

IVM formulae for bag union and monus

Table 7.3 illustrates IVM formulae for bag union and monus (derived from [GL95]), in which $\cap$ is an intersection operator with the following semantics: $\texttt{D1} \cap \texttt{D2} = \texttt{D1} -- (\texttt{D1} -- \texttt{D2}) = \texttt{D2} -- (\texttt{D2} -- \texttt{D1})$. The IVM formulae for bag union only use the changes to the data sources, while the formulae for bag monus have to use the view data and require an auxiliary view $\texttt{D2} -- \texttt{D1}$. This auxiliary view is similarly incrementally maintained by using the IVM formulae for bag monus with $\texttt{D1} -- \texttt{D2}$.

IVM formulae for comprehensions

We first discuss IVM formulae for a comprehension $[\overline{x}|\overline{x_1} \leftarrow \texttt{D1}; \ldots; \overline{x_n} \leftarrow \texttt{Dn}; C_1; C_2; ...; C_k]$ without member and not member expressions appearing in the filters.

For ease of discussion, we use the join operator $\bowtie$ to express this comprehension. In particular, $(\texttt{D1} \bowtie_c \texttt{D2}) = [\{x,y\}|x \leftarrow \texttt{D1}; y \leftarrow \texttt{D2}; c]$ where

v		IVM Formulae
$\mathtt{D1} +\!\!+ \mathtt{D2}$	$\triangle$v	$(\triangle\mathtt{D1} -- \nabla\mathtt{D2}) +\!\!+ (\triangle\mathtt{D2} -- \nabla\mathtt{D1})$
	∇v	$(\nabla\mathtt{D1} -- \triangle\mathtt{D2}) +\!\!+ (\nabla\mathtt{D2} -- \triangle\mathtt{D1})$
$\mathtt{D1} -- \mathtt{D2}$	$\triangle$v	$((\triangle\mathtt{D1} -- \triangle\mathtt{D2}) +\!\!+ (\nabla\mathtt{D2} -- \nabla\mathtt{D1})) -- (\mathtt{D2} -- \mathtt{D1})$
	∇v	$((\nabla\mathtt{D1} -- \nabla\mathtt{D2}) +\!\!+ (\triangle\mathtt{D2} -- \triangle\mathtt{D1})) \cap \mathtt{v}$

Table 7.3: IVM Formulae for Bag Union and Monus

$c = C_1; ...; C_k$. More generally, $(\mathtt{D1} \ltimes_{c_1,c_2} \mathtt{D2} \ltimes_{c_3} \ldots \ltimes_{c_n} \mathtt{Dn}) = [\overline{x} | \overline{x_1} \leftarrow \mathtt{D1}; \ldots; \overline{x_n} \leftarrow \mathtt{Dn}; c_1; c_2; ...; c_n]$ in which c_i is the conjunction of those predicates from $C_1, ..., C_k$ which contain variables appearing in $\overline{x_i}$ but without any variable appearing in $\overline{x_j}$, $j > i$.

We firstly give the IVM formulae of a view $\mathtt{v} = \mathtt{D1} \ltimes_c \mathtt{D2}$. The justification of these formulae is given in Appendix B.

$$\begin{aligned}
\triangle\mathtt{v} &= (\mathtt{D1}^{new} \ltimes_c \triangle\mathtt{D2} -- \triangle\mathtt{D1} \ltimes_c \triangle\mathtt{D2}) +\!\!+ \triangle\mathtt{D1} \ltimes_c \mathtt{D2}^{new} \\
\nabla\mathtt{v} &= (\nabla\mathtt{D1} \ltimes_c \mathtt{D2}^{new} -- \nabla\mathtt{D1} \ltimes_c \triangle\mathtt{D2}) +\!\!+ (\mathtt{D1}^{new} \ltimes_c \nabla\mathtt{D2} -- \triangle\mathtt{D1} \ltimes_c \nabla\mathtt{D2}) \\
&\quad +\!\!+ \nabla\mathtt{D1} \ltimes_c \nabla\mathtt{D2}
\end{aligned}$$

More generally, the IVM algorithm, IVM4Comp, for incrementally maintaining the view $\mathtt{v} = (\mathtt{D1} \ltimes_{c_1,c_2} \mathtt{D2} \ltimes_{c_3} \ldots \ltimes_{c_n} \mathtt{Dn})$ is given in Figure 7.1. This algorithm needs to access all the post-update data sources. It firstly computes the changes to the intermediate view $\mathtt{D1} \ltimes_{c_1,c_2} \mathtt{D2}$ based on the updates to the data source D1 and D2, and then checks the rest of data sources D3...Dn in turn. If there are updates to $\mathtt{D}_i$, a temporary view $\mathtt{tempView} = \mathtt{D1} \ltimes_{c_1,c_2} \mathtt{D2} \ltimes \ldots \ltimes_{c_{(i-1)}} \mathtt{D}_{(i-1)}$ is created in order to compute the changes to the intermediate view $\mathtt{D1} \ltimes_{c_1,c_2} \mathtt{D2} \ltimes \ldots \ltimes_{c_i} \mathtt{D}_i$. After checking all data sources of the view v, the changes to v have been computed.

The IVM4Comp algorithm is similar to the IVM algorithms discussed in references [ZGMW98] and [AASY97], *i.e.* the Strobe and SWEEP algorithms, in the context of maintaining a multi-source data warehouse. Both the Strobe and the SWEEP algorithm perform an IVM process for each update to a data source

```
Algorithm  IVM4Comp()
Begin:
           $\triangle$v = D1$^{new}$ $\bowtie_{c1,c2}$ $\triangle$D2 -- $\triangle$D1 $\bowtie_{c1,c2}$ $\triangle$D2) ++ $\triangle$D1 $\bowtie_{c1,c2}$ D2$^{new}$
           $\nabla$v = ($\nabla$D1 $\bowtie_{c1,c2}$ D2$^{new}$ -- $\nabla$D1 $\bowtie_{c1,c2}$ $\triangle$D2) ++ $\nabla$D1 $\bowtie_{c1,c2}$ $\nabla$D2
               ++(D1$^{new}$ $\bowtie_{c1,c2}$ $\nabla$D2 -- $\triangle$D1 $\bowtie_{c1,c2}$ $\nabla$D2)
           tempView = D1$^{new}$;
           for $i = 3$ to $n$, do
             if ($\triangle$D$i$ or $\nabla$D$i$ is not empty)
               tempView = tempView $\bowtie_{c_{(i-1)}}$ D$^{new}_{(i-1)}$;
               $\nabla$v = ($\nabla$v $\bowtie_{ci}$ D$i^{new}$ -- $\nabla$v $\bowtie_{ci}$ $\triangle$D$i$) ++ $\nabla$v $\bowtie_{ci}$ $\nabla$D$i$
                   ++(tempView $\bowtie_{ci}$ $\nabla$D$i$ -- $\triangle$v $\bowtie_{ci}$ $\nabla$D$i$);
               $\triangle$v = (tempView $\bowtie_{ci}$ $\triangle$D$i$ -- $\triangle$v $\bowtie_{ci}$ $\triangle$D$i$)++ $\triangle$v $\bowtie_{ci}$ D$i^{new}$;
             else
               $\triangle$v = $\triangle$v $\bowtie_{ci}$ D$i^{new}$;
               $\nabla$v = $\nabla$v $\bowtie_{ci}$ D$i^{new}$;
           return $\triangle$v and $\nabla$v;
End
```

Figure 7.1: The IVM4Comp Algorithm

so as to ensure the data warehouse is consistent with the updated data source. For both algorithms, the cost of the messaging between the data warehouse and the data sources for each update is $O(n)$ where n is the number of data sources. However, in practice, warehouse data are normally long-term and just refreshed periodically. Our IVM4Comp algorithm is able to handle a batch of updates and is specifically designed for a periodic view maintenance policy. The message cost of our algorithm for a batch of updates to any of the data sources is $O(n)$.

IVM formulae for `member` and `not member`

For ease of discussion, we use $\wedge$ and $\bar{\wedge}$ to denote expressions with `member` and `not member` operators, for example `D1` $\wedge$ `D2` denotes $[x|x \leftarrow$ `D1`; `member D2` $x]$ and `D1` $\bar{\wedge}$ `D2` denotes $[x|x \leftarrow$ `D1`; `not` (`member D2` x)$]$. The IVM formulae for `v` = [`x`|`x` $\leftarrow$ `D1`; `member D2 x`] are given below, in which the function `countNum a D` returns the number of occurrences of the data item `a` in `D`, *i.e.* `countNum a D` =

`count [x|x` $\leftarrow$ `D;x=a]`, and the priorities of $\wedge$ and $\overline{\wedge}$ operators are higher than $++$ and $--$ operators.

$$\begin{array}{lll}
\triangle\mathtt{v} & = & (\triangle\mathtt{D1} \wedge \mathtt{D2}^{new} -- \triangle\mathtt{D1} \wedge \mathtt{r1}) ++ \mathtt{D1}^{new} \wedge \mathtt{r1} \\
\nabla\mathtt{v} & = & (\mathtt{D1}^{new} \wedge \mathtt{r2} -- \triangle\mathtt{D1} \wedge \mathtt{r2}) ++ (\nabla\mathtt{D1} \wedge \mathtt{D2}^{new} -- \nabla\mathtt{D1} \wedge \mathtt{r1}) ++ \nabla\mathtt{D1} \wedge \mathtt{r2} \\
\text{where} & & \\
\mathtt{r1} & = & [x|x \leftarrow \triangle\mathtt{D2}; (\mathtt{countNum}\ x\ \ \triangle\mathtt{D2}) = (\mathtt{countNum}\ x\ \mathtt{D2}^{new})] \\
\mathtt{r2} & = & \nabla\mathtt{D2}\ \overline{\wedge}\ \mathtt{D2}^{new}
\end{array}$$

The IVM formulae for $\mathtt{v} = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D1}; \mathtt{not(member\ D2\ x)}]$ are as follows:

$$\begin{array}{lll}
\triangle\mathtt{v} & = & (\triangle\mathtt{D1}\ \overline{\wedge}\ \mathtt{D2}^{\mathtt{new}} -- \triangle\mathtt{D1} \wedge \mathtt{r2}) ++ \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} \\
\nabla\mathtt{v} & = & (\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} -- \triangle\mathtt{D1} \wedge \mathtt{r1}) ++ (\nabla\mathtt{D1}\ \overline{\wedge}\ \mathtt{D2}^{\mathtt{new}} -- \nabla\mathtt{D1} \wedge \mathtt{r2}) ++ \nabla\mathtt{D1} \wedge \mathtt{r1} \\
\text{where} & & \\
\mathtt{r1} & = & [x|x \leftarrow \triangle\mathtt{D2}; (\mathtt{countNum}\ x\ \ \triangle\mathtt{D2}) = (\mathtt{countNum}\ x\ \mathtt{D2}^{\mathtt{new}})] \\
\mathtt{r2} & = & \nabla\mathtt{D2}\ \overline{\wedge}\ \mathtt{D2}^{\mathtt{new}}
\end{array}$$

We can see that all post-update data sources are required in the IVM formulae. The justification of these formulae is given in Appendix *B*.

7.2.2 IVM over Schema Transformation Pathways

Having defined the IVM formulae for each kind of SIQL query, the update to a construct created by a single add transformation step is obtained by applying the appropriate formula to that step's query. Our IVM process for a single transformation step is IVM4AStep(cd, ts) and its output is the change to the construct created by transformation step ts based on the changes, cd, to ts's data sources.

As discussed above, the post-update data sources and the view itself are required by some IVM formulae. In a general transformation pathway, some intermediate constructs may be virtual. If a required data collection is unavailable, *i.e.* not materialised, the IVM4AStep procedure cannot be applied. Thus, we have to precheck each add transformation in the pathway. If a virtual data collection is required by the IVM formula for a transforation step, we must firstly materialise this data collection and store it in the data warehouse. This precheck only needs to be performed once for each transformation pathway, unless the transformation pathway evolves due to the evolution of a data source schema. This materialisation increases the storage overhead of the data warehouse, but does not increase the message cost of the IVM process since these materialised constructs are also maintainable by using the same IVM process along the transformation pathway.

Alternatively, we could use AutoMed's Global Query Processor (GQP) to evaluate the extent of a virtual construct during the IVM process so as to avoid increasing persistent storage overheads. However, since it uses post-update data sources, the GQP can only recover a post-update view. If a view itself is used in an IVM formula, *i.e.* the view *before* the update, this cannot be recovered by the GQP.

We now give an example of prechecking a transformation pathway. The transformation pathway generating $\langle\langle \mathsf{CourseSum}, \mathsf{Avg} \rangle\rangle$ in the global schema in Section 5.5.2 can be expressed as the following sequence of view definitions, where the intermediate constructs `v1`, ..., `v4` and $\langle\langle \mathsf{Details}, \mathsf{Mark} \rangle\rangle$ are virtual:

$$\begin{array}{lcl}
\texttt{v1} & = & [\texttt{\{'IS',k1,k2,x\}} \mid \texttt{\{k1,k2,x\}} \leftarrow \langle\langle \mathsf{IStab}, \mathsf{Mark} \rangle\rangle] \\
\texttt{v2} & = & [\texttt{\{'MA',k1,k2,x\}} \mid \texttt{\{k1,k2,x\}} \leftarrow \langle\langle \mathsf{MAtab}, \mathsf{Mark} \rangle\rangle] \\
\langle\langle \mathsf{Details}, \mathsf{Mark} \rangle\rangle & = & \texttt{v1} +\!\!+ \texttt{v2} \\
\texttt{v3} & = & \texttt{map}\ (\texttt{lambda \{k,k1,k2,x\}.\{\{k,k1\},x\}})\ \langle\langle \mathsf{Details}, \mathsf{Mark} \rangle\rangle \\
\texttt{v4} & = & \texttt{gc avg v3} \\
\langle\langle \mathsf{CourseSum}, \mathsf{Avg} \rangle\rangle & = & \texttt{map}\ (\texttt{lambda \{\{x,y\},z\}.\{x,y,z\}})\ \texttt{v4}
\end{array}$$

In order to incrementally maintain $\langle\langle \mathsf{CourseSum}, \mathsf{Avg} \rangle\rangle$, the intermediate views `v3` and `v4` must be materialised (based on the IVM formulae for grouping functions). For example, suppose that an update to the data sources is a tuple inserted into $\langle\langle \mathsf{IStab}, \mathsf{Mark} \rangle\rangle$, $\triangle\langle\langle \mathsf{IStab}, \mathsf{Mark} \rangle\rangle = $ `{'ISC01','ISS05',80}`. Following the transformation pathway, we obtain the changes to the intermediate views as follows:

$$\begin{aligned}
&\triangle\texttt{v1} &&= \texttt{\{'IS','ISC01','ISS05',80\}}\\
&\triangle\langle\langle\mathsf{Details},\mathsf{Mark}\rangle\rangle &&= \texttt{\{'IS','ISC01','ISS05',80\}}\\
&\triangle\texttt{v3} &&= \texttt{\{\{'IS','ISC01'\},80\}}
\end{aligned}$$

Since the extents of `v3` and `v4` are materialised, changes to `v4` can be obtained by using the IVM formulae for grouping functions, and then be used to compute changes to ⟨⟨CourseSum, Avg⟩⟩ by using the IVM formula for map expressions.

However, the post-update extent of `v3` can be recovered by AutoMed's GQP, and using the inverse query of `map (lambda {{x,y},z}.{x,y,z}) v4`, the pre-update extent of `v4` can also be recovered as `v4` = `map (lambda {x,y,z}.{{x,y}, z})` ⟨⟨CourseSum, Avg⟩⟩. Thus, in practice, no intermediate view needs to be materialised for incrementally maintaining ⟨⟨CourseSum, Avg⟩⟩ along the pathway.

7.3 Avoiding Materialisations in IVM

The above example shows that some materialisations in the IVM process are avoidable so reducing the storage overhead of a data warehouse. In this section, we will investigate these avoidable materialisations more generally, so as to apply them in our IVM process.

We consider five methods to avoid materialisations in our IVM process: using AutoMed's GQP; using view definitions; using inverse queries; IVM formulae for virtual schema constructs; and redefining view definitions. We now discuss these in turn in Section 7.3.1 – 7.3.5 below.

7.3.1 Using AutoMed's Global Query Processor

As described above, AutoMed's Global Query Processor (GQP) can be used to evaluate the extent of a virtual construct during the IVM process so as to avoid increasing persistent storage overheads. However, using the GQP will have higher time overheads than other methods discussed below since the GQP uses data source wrappers to access data sources for evaluating queries. Also it will require more memory than the other methods to store the result of the GQP evaluation. Furthermore, the GQP cannot be used to recover a view before the update since it uses post-update data sources.

7.3.2 Using View Definitions

Instead of using the GQP for recovering a virtual construct, we can use the view definition to replace the construct in our IVM formulae so that the query can be pushed to data sources to be evaluated rather than being evaluated by the GQP.

For example, the view definition of the virtual construct `v3` in Section 7.2.2 is as follows:

$$\begin{aligned}
\texttt{v3} &= \texttt{map (lambda \{k,k1,k2,x\}.\{\{k,k1\},x\})}\ \langle\langle \mathsf{Details}, \mathsf{Mark}\rangle\rangle \\
&= \texttt{map (lambda \{k,k1,k2,x\}.\{\{k,k1\},x\})} \\
&\qquad\qquad ([\texttt{\{'IS',k1,k2,x\}} \mid \texttt{\{k1,k2,x\}} \leftarrow \langle\langle \mathsf{IStab}, \mathsf{Mark}\rangle\rangle] ++ \\
&\qquad\qquad\ [\texttt{\{'MA',k1,k2,x\}} \mid \texttt{\{k1,k2,x\}} \leftarrow \langle\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\rangle]) \\
&= ([\texttt{\{\{'IS',k1\},x\}} \mid \texttt{\{k1,k2,x\}} \leftarrow \langle\langle \mathsf{IStab}, \mathsf{Mark}\rangle\rangle] ++ \\
&\quad\ [\texttt{\{\{'MA',k1\},x\}} \mid \texttt{\{k1,k2,x\}} \leftarrow \langle\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\rangle])
\end{aligned}$$

Then the IVM formula for computing $\triangle$`v4` can be transformed into:

$$\begin{aligned}
\triangle\texttt{v4} &= \texttt{gc avg } [\{x,y\}|\{x,y\} \leftarrow \texttt{v3}^{new};\\
&\qquad\qquad \texttt{member } [p|\{p,q\} \leftarrow (\triangle\texttt{v3} +\!+ \triangledown\texttt{v3})]\ x]\\
&= \texttt{gc avg } [\{x,y\}|\{x,y\} \leftarrow\\
&\quad ([\texttt{\{\{'IS',k1\},x\}|\{k1,k2,x\}} \leftarrow \langle\!\langle \mathsf{IStab}, \mathsf{Mark}\rangle\!\rangle]\\
&\quad +\!+[\texttt{\{\{'MA',k1\},x\}|\{k1,k2,x\}} \leftarrow \langle\!\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\!\rangle]);\\
&\qquad\qquad \texttt{member } [p|\{p,q\} \leftarrow (\triangle\texttt{v3} +\!+ \triangledown\texttt{v3})]\ x]\\
&= \texttt{gc avg}\\
&\quad ([\texttt{\{\{'IS',k1\},x\}|\{k1,k2,x\}} \leftarrow \langle\!\langle \mathsf{IStab}, \mathsf{Mark}\rangle\!\rangle;\\
&\qquad\qquad \texttt{member } [p|\{p,q\} \leftarrow (\triangle\texttt{v3} +\!+ \triangledown\texttt{v3})]\ \texttt{\{'IS',k1\}}]\\
&\quad +\!+[\texttt{\{\{'MA',k1\},x\}|\{k1,k2,x\}} \leftarrow \langle\!\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\!\rangle;\\
&\qquad\qquad \texttt{member } [p|\{p,q\} \leftarrow (\triangle\texttt{v3} +\!+ \triangledown\texttt{v3})]\ \texttt{\{'MA',k1\}}])
\end{aligned}$$

Thus, the two sub queries, $[\texttt{\{\{'IS',k1\},x\}|\{k1,k2,x\}} \leftarrow \langle\!\langle \mathsf{IStab}, \mathsf{Mark}\rangle\!\rangle; \texttt{member}\ [p|\{p,q\} \leftarrow (\triangle\texttt{v3} +\!+ \triangledown\texttt{v3})]\ \texttt{\{'IS',k1\}}]$ and $[\texttt{\{\{'MA',k1\},x\}|\{k1,k2,x\}} \leftarrow \langle\!\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\!\rangle; \texttt{member}\ [p|\{p,q\} \leftarrow (\triangle\texttt{v3}+\!+\triangledown\texttt{v3})]\ \texttt{\{'MA',k1\}}]$, can be pushed into the materialised data sources $\langle\!\langle \mathsf{IStab}, \mathsf{Mark}\rangle\!\rangle$ and $\langle\!\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\!\rangle$ respectively to be evaluated locally.

7.3.3 Using Inverse Queries

Some virtual intermediate schema constructs can be recovered from the constructs in the global schema using the *inverse query*, such as virtual construct `v4` in the example in Section 7.2.2. Suppose that `q` is an IQL^c query, and `v = q(D)`. If there is a query $\texttt{q}^{-1}$ such that $\texttt{D = q}^{-1}\texttt{(v)}$, we term $\texttt{q}^{-1}$ the *inverse query of* `q`.

The recovered constructs are pre-update ones since the inverse queries are based on the view constructs before the update. Thus, the approach of using inverse queries complements the approach of using AutoMed's GQP and view definitions which are based on post-update data sources.

However, not all queries have inverse queries. In SIQL, only `v = group D` always has an inverse query, `D = map (lambda {x,xl}.[{x,y}|{y} ← xl]) v`. The query `v = map (lambda p1.p2) D` also has an inverse query if and only if all variables appearing in `p1` are contained in `p2`: the corresponding inverse query is `D = map (lambda p2.p1) v`. Otherwise, `D` cannot be recovered from `v`.

7.3.4 IVM Formulae for Virtual Schema Constructs

We can develop IVM formulae for virtual schema constructs so as to avoid materialisations in our IVM process along AutoMed transformation pathways.

Considering a view `v` defined by a SIQL query `q` over data source `S`, `v = q(S)`, it is necessary that our IVM formulae can handle the following four cases: MvMs — both the view and the source data are materialised; MvVs — the view is materialised and the source data is virtual; VvMs — the view is virtual and the source data is materialised; and VvVs — both the view and the source data are virtual.

The IVM formulae for the case of MvMs were given in Section 7.2.1, and we now present the IVM formulae for the other three cases. Note that, we assume that updates to data sources and the update to the view are materialised.

Case MvVs

The IVM formulae for the case of MvMs given in Section 7.2.1 show that IVM formulae for `distinct`, `max`, `min`, `avg`, grouping functions and comprehensions are using the data sources. We now consider each of these kinds of SIQL queries. The IVM formulae for the other kinds of SIQL queries do not use the data sources as arguments, and thus do not need to be considered.

1. `v = distinct D`

   If `D` is virtual, $\triangledown$`v` is not obtainable if there are deletions, $\triangledown$`D`, from the data source.

2. `v = max/min D`

   If `D` is virtual, `v` is not maintainable if there are deletions, `▽D`, from the data source.

3. `v = avg D`

   If auxiliary views `v_s = sum D` and `v_c = count D` are available, `v` is maintained by following IVM formulae.

   $\triangle\texttt{v} = (\texttt{v\_s} + (\texttt{sum}\ \ \triangle\texttt{D}) - (\texttt{sum}\ \ \triangledown\texttt{D}))/(\texttt{v\_c} + (\texttt{count}\ \ \triangle\texttt{D}) - (\texttt{count}\ \ \triangledown\texttt{D}))$

   $\triangledown\texttt{v} = \texttt{v}$

4. `v = group D`

```
let r1  =  group  △D
    r2  =  group ▽D
    ▽v  =  [{x,y}|{x,y} ← v;
                    member (map (lambda {p,q}.p) (r1 ++ r2)) x]
let r3  =  [{x,y -- q}|{x,y} ← ▽v; {p,q} ← r2; x=p]
    r4  =  [{x,y ++ q}|{x,y} ← r3; {p,q} ← r1; x=p]
    △v  =  r4 ++ [{x,y}|{x,y} ← r1;
                          not (member (map (lambda {p,q}.p) r4) x)]
```

5. `v = gc max/min/avg D`

   `v` is not maintainable if `D` is virtual.

6. `v = gc sum/count D`

```
let r1  =  gc sum/count  △D
    r2  =  gc sum/count ▽D
    ▽v  =  [{x,y}|{x,y} ← v;
                    member (map (lambda {p,q}.p) (r1 ++ r2)) x]
```

```
let  r3  =  [{x,(y - q)}|{x,y} ← ∇v;{p,q} ← r2;x=p]
     r4  =  [{x,(y + q)}|{x,y} ← r3;{p,q} ← r1;x=p]
     △v  =  r4 ++ [{x,y}|{x,y} ← r1;
                           not (member (map (lambda {p,q}.p) r4) x)]
```

7. v is defined by comprehensions, including member and not member functions. If the data source is virtual, v is not maintainable.

Case VvMs

The IVM formulae for the case of MvMs, show that IVM formulae for `distinct`, aggregate functions, grouping functions and bag monus are using pre-update views. Here, we are not concerned with the situation of aggregate functions if the views are virtual, since the view of an aggregate function is a number which does not incur significant cost overheads. If such a materialised view is required for our IVM algorithms, we can store it in the data warehouse.

We now consider the IVM formulae for the SIQL queries listed above, except for aggregate functions, if the view is virtual but the source data is materialised:

1. `v = distinct D`
   △v = distinct [x|x ←△D; (countNum x D^{new}) = (countNum x △D)]

   ∇v = distinct [x|x ← ∇D; not (member D^{new} x)]

2. `v = group D`
   let r1 = [{x,y}|{x,y} ← D^{new}; member (map (lambda {p,q}.p) (△D ++ ∇D)) x]

   △v = group r1

   ∇v = group (r1 ++ ∇D −− △D)

3. `v = gc f D`

```
let  r1 = [{x,y}|{x,y} ← D^new; member (map (lambda {p,q}.p) (△D ++ ▽D)) x]
     r2 = group r1
     r3 = group (r1 ++ ▽D −− △D)
     r4 = r2 ∩ r3
     △v = r2 −− r4
     ▽v = r3 −− r4
```

4. `v = D1 −− D2`

   Suppose that `v` and the auxiliary view `D2 −− D1` are all unavailable.

```
let  r1 = △D1 ++ ▽D1++ △D2 ++ ▽D2
     r2 = [x|x ← D1^new; member r1 x]
     r3 = [x|x ← D2^new; member r1 x]
     r4 = r2 ++ ▽D1−− △D1
     r5 = r3 ++ ▽D2−− △D2
     r6 = r2 −− r3
     r7 = r4 −− r5
     △v = r6 −− r7
     ▽v = r7 −− r6
```

Case VvVs

In the case of VvVs, only views defined by `map` functions or ++ expressions are incrementally maintainable. The changes to the view are obtained from the updates to the data sources (see Section 7.2.1).

7.3.5 Redefining View Definitions

In our IVM process, materialisations may be avoided if we redefine the view definition. For example, suppose that `v = [x|x ← (D1 ++ D2); member D3 x]`, in which data sources `D1`, `D2` and `D3` are materialised. In order to incrementally maintain the view v, we decompose the view definition into the following SIQL queries, by using the rules for decomposing IQL^c queries given in Chapter 5:

$$\begin{aligned} \texttt{v1} &= \texttt{D1} +\!+ \texttt{D2} \\ \texttt{v} &= [\texttt{x|x} \leftarrow \texttt{v1}; \texttt{member D3 x}] \end{aligned}$$

Then, the intermediate view v1 must be materialised since it is a data source of a comprehension.

However, consider the view definition v' $= [\texttt{x|x} \leftarrow \texttt{D1}; \texttt{member D3 x}] +\!+ [\texttt{x|x} \leftarrow \texttt{D2}; \texttt{member D3 x}]$. Obviously, views v and v' are equivalent. The definition of $\texttt{v}'$ can be expressed as follows:

$$\begin{aligned} \texttt{v'} &= \texttt{v1'} +\!+ \texttt{v2'} \\ \texttt{v1'} &= [\texttt{x|x} \leftarrow \texttt{D1}; \texttt{member D3 x}] \\ \texttt{v2'} &= [\texttt{x|x} \leftarrow \texttt{D2}; \texttt{member D3 x}] \end{aligned}$$

We can see that no intermediate view is required to be materialised for computing the updates to the view v'.

The above example illustrates that if a comprehension contains ++ expressions as sub-queries, we can redefine the comprehension by pulling the ++ operators outside the comprehension, using the general equivalence $[\texttt{h|Q1}; \ldots; \overline{\texttt{x}_i} \leftarrow (\texttt{Di}_1 +\!+ \texttt{Di}_2); \ldots; \texttt{Qn}] = [\texttt{h|Q1}; \ldots; \overline{\texttt{x}_i} \leftarrow \texttt{Di}_1; \ldots; \texttt{Qn}] +\!+ [\texttt{h|Q1}; \ldots; \overline{\texttt{x}_i} \leftarrow \texttt{Di}_2; \ldots; \texttt{Qn}]$, so as to avoid materialising the intermediate results of these ++ expressions.

In practice, there two limitations of applying this kind of redefinition. One, if the source data of a ++ expression are virtual, for example $\texttt{Di}_1$ and $\texttt{Di}_2$ are virtual, applying the rule cannot save the storage overhead of materialisation. Since we have to either materialise the intermediate view $\texttt{Di}_1 +\!+ \texttt{Di}_2$, or materialise $\texttt{Di}_1$ and $\texttt{Di}_2$ individually.

Two, applying the rule will increase the number of comprehensions in a transformation pathway hence decreasing the efficiency of the IVM process. If the number of ++ expressions in a comprehension is n and the number of the data sources in each ++ expression is $a_i (1 \leq i \leq n)$, then the number of comprehensions created after applying the rule is $a_1 \times a_2 \times \ldots \times a_n$. From the IVM formulae for comprehensions given in Section 7.2.1, we can see that the time and temporary storage overheads of maintaining comprehensions are normally expensive, since we have to access each post-update source data and create temporary intermediate views if the number of generators in a comprehension is greater than 2. Thus, if the number of generators in a comprehension is greater than 2, we do not apply the redefinition rule.

7.4 Extending the IVM Algorithms

7.4.1 Using Queries beyond IQL^c

Our IVM algorithms above handle IQL^c queries in add transformations. However, add transformations for single-source cleansing may contain built-in functions which cannot be handled by our IVM formulae above. In order to maintain materialised single-cleansed schemas, the IVM process may therefore need to handle queries beyond IQL^c.

In particular, suppose the construct c is created by the following transformation step, in which f is a function defined by means of an arbitrary IQL query and $\mathtt{s_1}, ..., \mathtt{s_n}$ are the schemes appearing in the query:

addT(c, f($\mathtt{s_1}, ..., \mathtt{s_n}$));

We consider the IVM process propagating the changes to c, $\triangle \mathtt{c} / \triangledown \mathtt{c}$, according to the data source updates $\triangle \mathtt{s_1} / \triangledown \mathtt{s_1}, ..., \triangle \mathtt{s_n} / \triangledown \mathtt{s_n}$ in the following three cases:

1. f is an IQL^c query, in which case the DLT formulae described in this chapter can be used to compute $\triangle \mathtt{c} / \triangledown \mathtt{c}$;

2. $\mathtt{n} = \mathtt{1}$ and $\mathtt{f}$ is of the form $\mathtt{f(s_1)} = [\mathtt{h\ x}|\mathtt{x} \leftarrow \mathtt{s_1};\mathtt{C}]$ for some $\mathtt{h}$ and $\mathtt{C}$, in which case the changes to $\mathtt{c}$ are computed by the following formulae:

   $\triangle\mathtt{c} = [\mathtt{h\ x|x} \leftarrow \triangle\mathtt{s_1};\mathtt{C}]$

   $\nabla\mathtt{c} = [\mathtt{h\ x|x} \leftarrow \nabla\mathtt{s_1};\mathtt{C}]$

   More generally, if the following hold for `f`

   $\mathtt{f(S} +\!+ \mathtt{T)} = \mathtt{op\ f(s)\ f(T)}$

   $\mathtt{f(S} -\!- \mathtt{T)} = \mathtt{op'\ f(s)\ f(T)}$

   for some pair of operators `op` and `op'` such that `(a op b) op' b = a` for all `a,b` (e.g. if `op` = `+` and `op'` = `-`, or `op` = `++` and `op'` = `--`), then, we can incrementally compute `c` if $\mathtt{s_1}$ changes.

   In particular, if the operator `op` is $++$ and `op'` is $--$, the changes to `c` are given by:

   $\triangle\mathtt{c} = f(\triangle\mathtt{s_1})$

   $\nabla\mathtt{c} = f(\nabla\mathtt{s_1})$

   Otherwise, the new extent of `c`, $\mathtt{c^{new}}$, is incrementally computed by the following formula:

   $\mathtt{c^{new}} = \mathtt{op'\ (op\ c\ f(}\triangle\mathtt{s_1))\ f(}\nabla\mathtt{s_1)}$

   and the changes to `c` are given by:

   $\triangle\mathtt{c} = \mathtt{c^{new}} -\!- \mathtt{c}$

   $\nabla\mathtt{c} = \mathtt{c} -\!- \mathtt{c^{new}}$

3. For all other cases, the new extent of `c`, $\mathtt{c^{new}}$, is fully recomputed from scratch and the changes to `c` are given by:

   $\triangle\mathtt{c} = \mathtt{c^{new}} -\!- \mathtt{c}$

   $\nabla\mathtt{c} = \mathtt{c} -\!- \mathtt{c^{new}}$

7.4.2 Using extend transformations

So far, we have considered only add and rename transformations. In this section, we discuss how to utilise extend transformations in our IVM process.

We recall from Chapter 3 that an extend transformation is applied if the extent of a new construct cannot be precisely derived from the source schema. The transformation $\mathsf{extendT}(\mathsf{c}, ql, qu)$ adds a new construct c to a schema, where the query ql determines from the schema what is the minimum extent of c (and may be Void) and the query qu determines what is the maximal extent of c (and may be Any).

If the transformation is $\mathsf{extendT}(\mathsf{c}, \mathsf{Void}, \mathsf{Any})$, this means that no information about the extent of c can be derived from the source schema. We terminate the IVM process for computing changes to construct c at that step.

If the transformation is $\mathsf{extendT}(\mathsf{c}, ql, \mathsf{Any})$, this means the extent of c can be partially recovered by the query ql. Using ql, we can compute the changes, $\triangle\mathsf{c}/\triangledown\mathsf{c}$, to construct c. Since ql is a lower bound on the extent of c, we can insert $\triangle\mathsf{c}$ into c safely. However, we cannot simply delete $\triangledown\mathsf{c}$ from c, because c may contain more data than the result of ql. Similarly, if the transformation is $\mathsf{extendT}(\mathsf{c}, \mathsf{Void}, qu)$, the result of the query qu may contain more data than construct c. $\triangledown\mathsf{c}$ computed based on qu can be simply deleted from c, but $\triangle\mathsf{c}$ cannot be inserted into c safely.

Finally, if the transformation is $\mathsf{extendT}(\mathsf{c}, ql, qu)$, we firstly compute the changes to c based on these two queries, and select $\triangle\mathsf{c}$ based on ql and $\triangledown\mathsf{c}$ based on qu to update c. However, we have to indicate to the data warehouse users that such updates may not be the exact changes to the view construct c.

7.5 Discussion

AutoMed schema transformation pathways can be used to express data transformation and integration processes in heterogeneous data warehousing environments. This chapter has discussed techniques for incremental view maintenance along such pathways. We have developed a set of IVM formulae. Based on these formulae, our algorithms perform an IVM process along a schema transformation pathway. We also have discussed approaches for avoiding materialisations in our IVM algorithms so as to save storage overheads.

One of the advantages of AutoMed is that its schema transformation pathways can be readily evolved as the data warehouse evolves. In this chapter we have shown how to perform IVM along such evolvable pathways.

Although this chapter has used IQL^c as the query language in which transformations are specified, our algorithms are not limited to one specific data model or query language, and could be applied to other query languages involving common algebraic operations on collections such as selection, projection, join, aggregation, union and difference.

Finally, since our algorithms consider in turn each transformation step in a transformation pathway in order to compute data changes in a stepwise fashion, they are useful not only in data warehousing environments, but also in any data transformation and integration framework based on sequences of primitive schema transformations, such as peer-to-peer and semi-structured data integration environments.

Chapter 8

Conclusions and Future Work

This thesis has discussed the use of the both-as-view (BAV) data integration approach and the AutoMed toolkit for data warehousing. There are three main advantages in using BAV and AutoMed for data warehousing: (*i*) the data source wrappers translate each data source schema into its equivalent AutoMed representation; any necessary inter-model translation then happens explicitly within the AutoMed transformation pathways, under the control of the data warehouse designer; (*ii*) if the data warehouse is to be redeployed on a platform with a different data model, it is easy to reuse the previous data transformation and implementation effort; (*iii*) evolutions of the data source schemas and the data warehouse schema are readily supported. Point (*i*) was discussed in Chapter 3 of this thesis, and points (*ii*) and (*iii*) were discussed in Chapter 4.

In order to use AutoMed for heterogenous data warehousing, we considered the following four research problems in this thesis: how AutoMed metadata can be used to express the schemas of a data warehouse and processes such as data cleansing, transformation and integration; how schema evolution can be handled; how AutoMed metadata can be used for data lineage tracing; and how AutoMed metadata can be used for incremental view maintenance.

Our solutions to these problems are in the context of a heterogeneous data warehouse environment where evolutions of the data source schemas and the data warehouse schema may occur, including changes in the data models in which these schemas have been represented.

In Chapter 2, we have given an overview of the major issues in data warehousing, which include the definition of a data warehouse, data warehouse architecture, data warehouse modelling, and data warehouse processes.

In Chapter 3, we have discussed how AutoMed metadata can be used in a data warehousing environment. We have shown how AutoMed metadata can be used to express the schemas of the data sources and of the data warehouse, and to represent data warehouse processes such as data cleansing, transformation, integration, summarisation and creating data marts.

In Chapter 4, we have described how AutoMed schema transformations can be used to express the evolution of schemas in a data warehouse. We have shown how the existing warehouse metadata and data can be evolved so that the previous transformation, integration and data materialisation effort can be reused.

In Chapters 5 and 6, we have addressed the problem of data lineage tracing (DLT), *i.e.* finding the derivation in the data sources of the tracing data in the global database. In particular, Chapter 5 has given the definitions of data lineage in the context of AutoMed, presented a method for tracing data lineage along a materialised AutoMed transformation pathway and discussed the problem of derivation ambiguity in data lineage tracing. Chapter 6 has then generalised the DLT algorithms to handle virtual intermediate transformation steps, so that our DLT process can be applied along a general transformation pathway. The main contributions of our DLT approach are as follows:

Firstly, we have considered both why- and where-provenance using bag semantics and have given the definition of affect-pool and origin-pool for data lineage in the context of AutoMed. In contrast, the previous work of Cui *et al* only considered why-provenance.

Secondly, we have developed a set of DLT formulae using virtual arguments to handle virtual intermediate schema constructs and virtual lineage data. Based on these formulae, we have presented algorithms which perform data lineage tracing along a general schema transformation pathway.

In practice, we use virtual lineage data to express the intermediate lineage data even if it is available. This can save in time and memory usage of the DLT process, and makes our DLT process applicable in both materialised and virtual data integration scenarios.

Although we have used IQL^c as the query language in which transformations are specified, our algorithms are not limited to one specific data model or query language, and could be applied to other query languages involving common algebraic operations on collections such as selection, projection, join, aggregation, union and difference.

Thirdly, since our algorithms consider in turn each transformation step in a transformation pathway in order to evaluate lineage data in a stepwise fashion, they are useful not only in data warehousing environments, but also in any data transformation and integration framework based on sequences of primitive schema transformations.

In Chapter 7, we have developed a set of incremental view maintenance (IVM) formulae. Based on these formulae, we have presented algorithms which perform an IVM process along a schema transformation pathway. We have also discussed approaches for avoiding materialisations in our IVM algorithms so as to reduce storage overheads.

The major results of Chapter 3 have been published in [FP03b] and those of Chapter 4 in [FP04]. The DLT algorithm of Chapter 5 has been published in

[FP02, FP03a] and that of Chapter 6 in [FP05]. The major results of Chapter 7 have been published in [Fan05].

Although developed in the context of AutoMed and a data warehousing environment, the techniques described in this thesis can be applied in any materialised data integration environment in which the data transformation and integration logic is expressed by sequences of schema transformations. This approach is likely to be beneficial in situations involving data transformation and integration across multiple data models and where both source and integrated schemas may frequently evolve. Grid, peer-to-peer and semi-structured data integration environments are likely to have these characteristics because they involve heterogeneous, distributed, autonomous data sources which are accessed and integrated across a network. Both the metadata and the data of these data sources may autonomously evolve. Also, different integrated schemas will be needed to meet the needs of different end-users and applications, and these integrated schemas may be dynamic and evolving e.g. new schemas created for new user requirements and existing schemas changed for updated user requirements.

In more static and homogeneous data integration environments, traditional approaches using one common data model with GAV or LAV views are likely to be more appropriate because they have simpler metadata to manage — just one common data model, and a set of view definitions rather than a set of schema transformation pathways. Also, if there is not a requirement to support frequent schema evolutions, processes such as global query evaluation, populating integrated schemas and maintaining materialised views may be more efficient using a set of view definitions directly compared with using a set of schema transformation pathways.

We are currently pursuing several directions of research building on the results of this thesis:

1. Implementation of data warehouse maintenance

   Materialised data warehouse views need to be maintained when the data sources change, and much previous work has addressed this problem at the data level, as did this thesis in Chapter 7. However, as discussed in Chapter 4, materialised views may also need to be modified if there is an evolution of a data source schema. We have discussed methods for handling such schema evolutions in that chapter. We now need to develop detailed algorithms. We will then combine our view maintenance approaches at the data level (from Chapter 7) and at the schema level (from Chapter 4), in order to develop a toolkit to handle the general view maintenance problem of a data warehouse.

2. Extension of our DLT & IVM approaches

   The DLT and IVM approaches described in this thesis assume IQL^c as the query language. However, our approaches can be easily modified to handle other query languages involving common algebraic operations on collections such as selection, projection, join, aggregation, union and difference. Furthermore, our DLT and IVM approaches are both performed in a step-wise fashion, and so any data transformation and integration framework based on sequences of schema transformations can use these approaches, *e.g.* [SKR01, YLT03]. In particular, we wish to extend our approaches to handle multiple query languages and to apply to web-based data integration environments.

3. Extension to peer-to-peer environments

   So far, we have assumed a single global schema for the DLT and IVM approaches described in this thesis. However, AutoMed can also be used in peer-to-peer data integration settings [MP03b]. We plan to extend our DLT and IVM algorithms to be applicable in peer-to-peer environments.

4. Application in biological data integration

   It is planned to apply the results of this thesis in the ongoing projects BioMap[1] and ISPIDER[2]. BioMap is developing a warehouse integrating protein family, structure, function and pathway/process data with gene expression and other experimental data, which aims to provide an integrated sequence/structure/function resource that supports analysis, mining and visualisation of functional genomics data. ISPIDER aims to provide an integrated platform of proteomic data resources enabled as Grid and Web services for the storage, dissemination and management of proteomic data, and to produce appropriate middleware technologies for distributed querying, workflows and other integrated data analysis tasks across this range of proteome databases.

   Reference [MZR$^+$05] gives an initial discussion of how the AutoMed toolkit can be used for integrating heterogeneous biological data sources, both for materialised integration as in BioMap and for virtual integration as in ISPIDER. Biological data sources typically have a very high degree of heterogeneity in terms of the type of data model used, the schema design within a given data model, as well as incompatible formats and naming of values. Reference [MZR$^+$05] identifies that the particular strengths of using AutoMed for biological data integration are that it supports reversible, extensible transformations from data source schemas to an integrated schema, and enables both virtual and materialised integration.

[1]See `http://www.biochem.ucl.ac.uk/bsm/biomap/index.html`
[2]See `http://www.ispider.man.ac.uk/`

It is expected that the results of this thesis, and also extensions 1-3 above, will benefit the above two projects by enabling incremental view maintenance for the BioMap warehouse and by enabling data lineage tracing for both BioMap and ISPIDER. Moreover, this will be in a context where evolutions of the data source schemas and the integrated schemas are readily supported, thus accommodating future changes of the BioMap and ISPIDER data sources and of their integrated schemas.

Bibliography

[AASY97] Divyakant Agrawal, Amr El Abbadi, Ambuj K. Singh, and Tolga Yurek. Efficient view maintenance at data warehouses. In *Proc. of ACM SIGMOD'97*, pages 417–427. ACM Press, 1997.

[AFP03] M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. MOVIE: An incremental maintenance system for materialized object views. *Data & Knowledge Engineering*, 47(2):131–166, 2003.

[Alb91] J. Albert. Algebraic properties of bag data types. In *Proc. of International Conference on Very Large Data Bases (VLDB'91)*, pages 211–219. Morgan Kaufmann, 1991.

[ALP91] J. Andany, M. Léonard, and C. Palisser. Management of schema evolution in databases. In *Proc. of International Conference on Very Large Data Bases (VLDB'91)*, pages 161–170. Morgan Kaufmann, 1991.

[AMGF05] M. B. Al-Mourad, W. Alex Gray, and N. Fiddian. Semantically rich materialisation rules for integrating heterogeneous databases. In *Proc. of British National Conference on Databases (BNCOD'05), LNCS 3567*, pages 60–69, 2005.

[BB99] P. A. Bernstein and T. Bergstraesser. Meta-data support for data transformations using microsoft repository. *IEEE Data Engineering Bulletin*, 22(1):9–14, 1999.

[BCDS01] Angela Bonifati, Fabio Casati, Umeshwar Dayal, and Ming-Chien Shan. Warehousing workflow data: Challenges and opportunities. In Proc. of International Conference on Very Large Data Bases (VLDB'01), pages 649–652, 2001.

[BCL89] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems (TODS)*, 14(3):369–400, 1989.

[BCRP98] G.S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.

[Bek99] Lars Bekgaard. Event-Entity-Relationship modeling in data warehouse environments. In *Proc. of International Workshop on Data Warehousing and OLAP (DOLAP'99)*, pages 9–14. ACM, 1999.

[Bel96] Z. Bellahsene. View mechanism for schema evolution in object-oriented DBMS. In *Proc. of British National Conference on Databases (BNCOD'96), LNCS 1094*, pages 18–35, Springer, 1996.

[Ben99] B. Benatallah. A unified framework for supporting dynamic schema evolution in object databases. In *Proc. of ER'99, LNCS 1728*, pages 16–30, Springer, 1999.

[BGF02] M. Burgess, W. Alex Gray, and N. Fiddian. Establishing a taxonomy of quality for use in information filtering. In *Proc. of British National Conference on Databases (BNCOD'02), LNCS 2405*, pages 103–113, 2002.

[BIG94] J. M. Blanco, A. Illarramendi, and A. Goñi. Building a federated database system: An approach using a knowledge base system. *International Journal of Intelligent and Cooperative Information Systems*, 3(4):415–455, 1994.

[BKL$^+$04] M. Boyd, S. Kittivoravitkul, C. Lazanitis, P.J. McBrien, and N. Rizopoulos. AutoMed: A BAV data integration system for heterogeneous data sources. In *Proc. of International Conference on Advanced Information Systems Engineering (CAiSE'04), LNCS 3084*, pages 82–97, Springer-Verlag, 2004.

[BKT00] P. Buneman, S. Khanna, and W.C. Tan. Data provenance: some basic issues. In *Proc. of 20th Conference in Foundations of Software Technology and Theoretical Computer Science, (FST TCS) New Delhi, India, LNCS 1974*, pages 87–93. Springer, 2000.

[BKT01] P. Buneman, S. Khanna, and W.C. Tan. Why and Where: A characterization of data provenance. In *Proc. of 8th International Conference in Database Theory - ICDT'01, London, UK*, *LNCS 1973*, pages 316–330. Springer, 2001.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In Carlo Zaniolo, editor, *Proc. of ACM SIGMOD'86*, pages 61–71. ACM Press, 1986.

[BMT02] M. Boyd, P. McBrien, and N. Tong. The AutoMed schema integration repository. In *Proc. of British National Conference on Databases (BNCOD'02), LNCS 2405*, pages 42–45. Springer, 2002.

[BSH99] M. Blaschka, C. Sapia, and G. Höfling. On schema evolution in multidimensional databases. In *Proc. of Data Warehousing and Knowledge Discovery (DaWaK'99), LNCS 1767*, pages 153–164, Springer, 1999.

[BTM01] Nguyen Thanh Binh, A. Min Tjoa, and Oscar Mangisengi. Meta Cube-X: An XML metadata foundation for interoperability search among web data warehouses. In *Proc. of Design and Management of Data Warehouses (DMDW'01)*, page 8, 2001.

[Bun94] P. Buneman *et al.* Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[CB02] Liane Carneiro and Angelo Brayner. X-META: A methodology for data warehouse design with metadata management. In *Proc. of Design and Management of Data Warehouses (DMDW'02)*, pages 13–22, 2002.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[CEM01] L. Capra, W. Emmerich, and C. Mascolo. Reflective middleware solutions for context-aware applications. In *Proc. of Metalevel Architectures and Separation of Crosscutting Concerns, Third International Conference, REFLECTION 2001*, *LNCS 2192*, pages 126–133. Springer, 2001.

[CGL$^+$96] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. of ACM SIGMOD'96*, pages 469–480, 1996.

[CGL$^+$99] D. Calvanese, G. D. Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. A principled approach to data integration and reconciliation in data warehousing. In *Proc. of Design and Management of Data Warehouses (DMDW'99)*, page 16, 1999.

[Cui01] Y. Cui. *Lineage tracing in data warehouses.* PhD thesis, Computer Science Department, Stanford University, 2001.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of International Conference on Very Large Data Bases (VLDB'91)*, pages 577–589. Morgan Kaufmann, 1991.

[CW00a] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *Proc. of International Conference on Data Engineering (ICDE'00)*, pages 367–378. IEEE Computer Society, 2000.

[CW00b] Y. Cui and J. Widom. Storing auxiliary data for efficient maintenance and lineage tracing of complex views. In *Proc. of Design and Management of Data Warehouses (DMDW'00)*, page 11, 2000.

[CW01] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *Proc. of International Conference on Very Large Data Bases (VLDB'01)*, pages 471–480. Morgan Kaufmann, 2001.

[CWW00] Y. Cui, J. Widom, and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.

[Don99] Guozhu Dong. Incremental maintenance of recursive views: A survey. In A Gupta and I. S. Mumick, editors, *Materialized Views: Techniques, Implementations, and Applications*, pages 159–162. The MIT Press, London, 1999.

[DZR99] Lingli Ding, Xin Zhang, and Elke A. Rundensteiner. The mre wrapper approach: Enabling incremental view maintenance of data warehouses defined on multi-relation information sources. In *Proc. of International Workshop on Data Warehousing and OLAP (DOLAP'99)*, pages 30–35. ACM, 1999.

[ECL03] H. Engstrom, S. Chakravarthy, and B. Lings. Maintenance policy selection in heterogeneous data warehouse environments: a heuristics-based approach. In *Proc. of International Workshop on Data Warehousing and OLAP (DOLAP'03)*, pages 71–78. ACM Press, 2003.

[Emm00] W. Emmerich. Software engineering and middleware: A roadmap. In *Proc. of 22th International Conference on Software Engineering (ICSE2000)*, pages 117–129. ACM Press, 2000.

[Eng02] Henrik Engstrom. *Selection of Maintenance Policies for a Data Warehousing Environment*. PhD thesis, University of Exeter, 2002.

[Fan05] H. Fan. Using schema transformation pathways for incremental view maintenance. In *Proc. of Data Warehousing and Knowledge Discovery (DaWaK'05), LNCS 3589*, pages 126–135, 2005.

[FJS97] C. Faloutsos, H.V. Jagadish, and N.D. Sidiropoulos. Recovering information from summary data. In *Proc. of International Conference on Very Large Data Bases (VLDB'97)*, pages 36–45. Morgan Kaufmann, 1997.

[FKP04] R. Fagin, P.G. Kolaitis, L. Popa, and W.C. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. In *Proc. of ACM Symposium on Principles of Database Systems (PODS'04)*, pages 83–94, ACM, 2004.

[FP02] H. Fan and A. Poulovassilis. Tracing data lineage using schema transformation pathways. In *Proc. of Workshop on Knowledge Transformation for the Semantic Web (with ECAI'02), Lyon*, 2002.

[FP03a] H. Fan and A. Poulovassilis. Tracing data lineage using schema transformation pathways. In B.Omelayenko and M.Klein, editors, *Knowledge Transformation for the Semantic Web*, volume 95 of *Frontiers in Artificial Intelligence and Applications*, pages 64–79. IOS Press, 2003.

[FP03b] H. Fan and A. Poulovassilis. Using AutoMed metadata in data warehousing environments. In *Proc. of International Workshop on Data Warehousing and OLAP (DOLAP'03)*, pages 86–93. ACM Press, 2003.

[FP04] H. Fan and A. Poulovassilis. Schema evolution in data warehousing environments — a schema transformation-based approach. In *Proc. of International Conference on Conceptual Modeling (ER'04)*, volume 3288 of *LNCS*, pages 639–653, Springer, 2004.

[FP05] H. Fan and A. Poulovassilis. Using schema transformation pathways for data lineage tracing. In *Proc. of British National Conference on Databases (BNCOD'05), LNCS 3567*, pages 133–144, 2005.

[GFS$^+$01a] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.A. Saita. Declarative data cleaning: Language, model, and algorithms. In *Proc. of International Conference on Very Large Data Bases (VLDB'01)*, pages 371–380. Morgan Kaufmann, 2001.

[GFS$^+$01b] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.A. Saita. Improving data cleaning quality using a data lineage facility. In *Proc. of Design and Management of Data Warehouses (DMDW'01)*, page 3, 2001.

[GFSS00] Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. AJAX: An extensible data cleaning tool. In *Proc. of ACM SIGMOD'00*, volume 29, page 590. ACM, 2000.

[GGMS97] D Gluche, T. Grust, C. Mainberger, and M. H. Scholl. Incremental updates for materialized oql views. In *Proc. of International Conference on Deductive and Object-Oriented Databases (DOOD'97)*, pages 52–66. Springer, 1997.

[GJM96] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *Extending Database Technology*, pages 140–144, 1996.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. of ACM SIGMOD'95*, pages 328–339. ACM Press, 1995.

[GM99] Ashish Gupta and Inderpal Singh Mumick. Maintenance polices. In A Gupta and I. S. Mumick, editors, *Materialized Views: Techniques, Implementations, and Applications*, pages 9–11. The MIT Press, London, 1999.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. of ACM SIGMOD'93*, pages 157–166. ACM Press, 1993.

[GR98] M. Golfarelli and S. Rizzi. A methodological framework for data warehouse design. In *Proc. of International Workshop on Data Warehousing and OLAP (DOLAP'98)*, pages 3–9. ACM, 1998.

[HA01] H. Hinrichs and T. Aden. An ISO 9001: 2000 compliant quality management system for data integration in data warehouse systems. In *Proc. of Design and Management of Data Warehouses (DMDW'01)*, page 1, 2001.

[HLV00] Bodo Hüsemann, Jens Lechtenbörger, and Gottfried Vossen. Conceptual data warehouse modeling. In *Proc. of Design and Management of Data Warehouses (DMDW'00)*, page 6, 2000.

[HMT00] Thanh N. Huynh, Oscar Mangisengi, and A. Min Tjoa. Metadata for object-relational data warehouse. In *Proc. of Design and Management of Data Warehouses (DMDW'00)*, page 3, 2000.

[HQGW93] N. I. Hachem, K. Qiu, M. A. Gennert, and M. O. Ward. Managing derived data in the Gaeas scientific DBMS. In *Proc. of International Conference on Very Large Data Bases (VLDB'93)*, pages 1–12. Morgan Kaufmann, 1993.

[Huy97] Nam Huyn. Multiple-view self-maintenance in data warehousing environments. In *Proc. of International Conference on Very Large Data Bases (VLDB'97)*, pages 26–35. Morgan Kaufmann, 1997.

[Inm02] W. H. Inmon. *Building The Data Warehouse.* John Wiley & Sons, third edition, March 2002.

[JPZ03] E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL queries and migrating data in the AutoMed toolkit. Technical Report 20, Automed Project, 2003.

[JTMP04] E. Jasper, N. Tong, P. McBrien, and A. Poulovassilis. View generation and optimisation in the AutoMed data integration framework. In *Proc. of 6th Baltic Conference on Databases and Information Systems*, 2004.

[KLM$^+$97] Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, Dallan Quass, and Kenneth A. Ross. Concurrency control theory for deferred materialized views. In *Proc. of International Conference on Database Theory - ICDT '97*, *LNCS 1186*, pages 306–320. Springer, 1997.

[KR02] A. Koeller and E. A. Rundensteiner. Incremental maintenance of schema-restructuring views. In *Proc. of EDBT'02, LNCS 2287*, pages 354–371, Springer, 2002.

[Lan02] Paul Lane. *Oracle9i Data Warehousing Guide, Release 2(9.2).* Oracle Corporation, March 2002.

[Len02] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. of ACM Symposium on Principles of Database Systems (PODS'02)*, pages 233–246, ACM, 2002.

[Lev00] A. Levy. Answering queries using views: A survey. In *The VLDB Journal*, 10(4), pages 270–294, 2001.

[LLL00] Mong-Li Lee, Tok Wang Ling, and Wai Lup Low. Intelliclean: a knowledge-based intelligent data cleaner. In *Knowledge Discovery and Data Mining*, pages 290–294, 2000.

[LLL01] Wai Lup Low, Mong Li Lee, and Tok Wang Ling. A knowledge-based framework for duplicates elimination. *Information Systems: Special Issue on Data Extraction, Cleaning and Reconciliation*, 28(8), December 2001. Elsevier Science.

[LLWO99] W. Liang, H. Li, H. Wang, and M. E. Orlowska. Making multiple views self-maintainable in a data warehouse. *Data & Knowledge Engineering*, 30(2):121–134, 1999.

[LNE89] J.A. Larson, S.B. Navathe, and R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE Transcations on Softerware Engineering*, 15(4):449–463, 1989.

[LSS93] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. In *Proc. of the Third International Conference in Deductive and Object-Oriented Databases (DOOD'93), LNCS 760*, pages 81–100, Springer-Verlag, 1993.

[LSS99] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. On efficiently implementing SchemaSQL on an SQL database system. In *Proc. of International Conference on Very Large Data Bases (VLDB'99)*, pages 471–482. Morgan Kaufmann, 1999.

[LSS01] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. Schemasql: An extension to sql for multidatabase interoperability. In *ACM Transactions on Database Systems (TODS), Volume 26 , Issue 4*, pages 476 – 519. ACM Press, 2001.

[MH03] J. Madhavan and A.Y. Halevy. Composing mappings among data sources. In *Proc. of International Conference on Very Large Data Bases (VLDB'03)*, pages 572–583, Morgan Kaufmann, 2003.

[Mil98] Renée J. Miller. Using schematically heterogeneous structures. In *Proc. of ACM SIGMOD'98*, pages 189–200. ACM Press, 1998.

[MK00] D. L. Moody and M. A. R. Kortink. From enterprise models to dimensional models: a methodology for data warehouse and data mart design. In *Proc. of Design and Management of Data Warehouses (DMDW'00)*, page 5, 2000.

[MP98] P. McBrien and A. Poulovassilis. A formalisation of semantic schema integration. *Information Systems*, 23(5):304–334, 1998.

[MP99a] P. McBrien and A. Poulovassilis. Automatic migration and wrapping of database applications - a schema transformation approach. In *Proc. of International Conference on Conceptual Modeling (ER'99)*, *LNCS 1728*, pages 96–113. Springer, 1999.

[MP99b] P. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Proc. of International Conference on Advanced Information Systems Engineering (CAiSE'99), LNCS 1626*, pages 333–348. Springer, 1999.

[MP01] P. McBrien and A. Poulovassilis. A semantic approach to integrating XML and structured data sources. In *Proc. of International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *LNCS*, pages 330–345. Springer, 2001.

[MP02] P. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proc. of International Conference on Advanced Information Systems Engineering (CAiSE'02)*, *LNCS 2348*, pages 484–499. Springer, 2002.

[MP03a] P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. of International Conference on Data Engineering (ICDE'03)*, pages 227–238, IEEE Computer Society, 2003.

[MP03b] P. McBrien and A. Poulovassilis. Defining peer-to-peer data integration using both as view rules. In *Proc. of Databases, Information Systems, and Peer-to-Peer Computing International Workshop (DBISP2P)*, *LNCS 2944*, pages 91–107, Springer, 2003.

[MS01] G. Moro and C. Sartori. Incremental maintenance of multi-source views. In *Proc. of Australasian Database Conference (ADC'01)*, pages 13–20, 2001.

[MSR99] Robert Müller, Thomas Stöhr, and Erhard Rahm. An integrative and uniform model for metadata management in data warehousing environments. In *Proc. of Design and Management of Data Warehouses (DMDW'99)*, page 12, 1999.

[MZR$^+$05] M. Maibaum, L. Zamboulis, G. Rimon, C. Orengo, N. Martin, and A. Poulovassilis. Cluster based integration of heterogeneous biological databases using the AutoMed toolkit. In *Proc. of Data Integration in the Life Sciences (DILS'05), LNCS 3615*, pages 191–207, 2005.

[PM98] A. Poulovassilis and P. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28(1):47–71, 1998.

[Pou04] A. Poulovassilis. A Tutorial on the IQL Query Language. Technical Report 28, Automed Project, 2004.

[PS97] A. Poulovassilis and C. Small. Formal foundations for optimising aggregation functions in database programming languages. In *Proc. of Database Programming Languages, International Workshop (DBPL'97), Springer-Verlag LNCS 1369*, pages 299–318, 1997.

[PSCP02] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proc. of International Conference on Very Large Data Bases (VLDB'02), LNCS 2590*, pages 802-813, 2002.

[QGMW96] D. Quass, A. Gupta, I.S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. of Conference* *on Parallel and Distributed Information Systems (PDIS'96)*, pages 158–169, 1996.

[Qua96] D. Quass. Maintenance expressions for views with aggregation. In *Proc of Workshop on Materialized Views: Techniques and Applications (VIEW'96)*, pages 110–118, 1996.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *Knowledge and Data Engineering*, 3(3):337–341, 1991.

[QW97] Dallan Quass and Jennifer Widom. On-line warehouse view maintenance. In Joan Peckham, editor, *Proc ACM SIGMOD'97*, pages 393–404. ACM Press, 1997.

[RD00] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[RH01] Vijayshankar Raman and Joseph M. Hellerstein. Potter's Wheel: An interactive data cleaning system. In *The VLDB Journal*, pages 381–390, 2001.

[Riz04] N. Rizopoulos. Automatic discovery of semantic relationships between schema elements. In *Proc. of International Conference on Enterprise Information Systems (ICEIS'04)*, pages 3–8, 2004.

[SG97] L. Serafini and C. Ghidini. Context based semantics for information integration. In Sasa Buvač and Łucia Iwańska, editors, *Working Papers of the AAAI Fall Symposium on Context in Knowledge*

Representation and Natural Language, pages 152–160, Menlo Park, California, 1997. American Association for Artificial Intelligence.

[SKR01] H. Su, H. Kuno, and E. A. Rudensteiner. Automating the transformation of XML documents. In *Proc. of International Workshop on Web Information and Data Management (WIDM'01)*, pages 68–75. ACM, 2001.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[TBC99] Nectaria Tryfona, Frank Busborg, and Jens G. Borch Christiansen. starER: A conceptual model for data warehouse design. In *Proc. of International Workshop on Data Warehousing and OLAP (DOLAP'99)*, pages 3–8. ACM, 1999.

[The02] D. Theodoratos. Semantic integration and querying of heterogeneous data sources using a hypergraph data model. In *Proc. of British National Conference on Databases (BNCOD'02), LNCS 2405*, pages 166–182, 2002.

[TKS01] A. Tsois, N. Karayannidis, and T. K. Sellis. MAC: Conceptual data modeling for OLAP. In *Proc. of Design and Management of Data Warehouses (DMDW'01)*, page 5, 2001.

[Ton03] N. Tong. Database schema transformation optimisation techniques for the AutoMed system. In *Proc. of British National Conference on Databases (BNCOD'03), LNCS 2712*, pages 157–171, Springer, 2003.

[VM97] M. W. Vincent and M. Mohania. A self-maintainable view maintenance technique for data warehouses. In *Proc. of International Conference on Management of Data (COMAD'97)*, pages 7–22, 1997.

[VMP03] Y. Velegrakis, R.J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *Proc. of International Conference on Very Large Data Bases (VLDB'03)*, pages 584-595, Morgan Kaufmann, 2003.

[VSS02] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Conceptual modeling for ETL processes. In *Proc. of International Workshop on Data Warehousing and OLAP (DOLAP'02)*, pages 14–21. ACM, 2002.

[VVSK00] Panos Vassiliadis, Zografoula Vagena, Spiros Skiadopoulos, and Nikos Karayannidis. ARKTOS: A tool for data cleaning and transformation in data warehouse environments. *IEEE Data Engineering Bulletin*, 23(4):42–47, 2000.

[Wid95] Jennifer Widom. Research problems in data warehousing. In *Proc. of CIKM '95, the International Conference on Information and Knowledge Management, Baltimore, Maryland, USA*, pages 25–30. ACM, 1995.

[WS97] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc. of International Conference on Data Engineering (ICDE'97)*, pages 91–102. IEEE Computer Society, 1997.

[YLT03] X. Yang, M.L. Lee, and T.W.Ling. Resolving structural conflicts in the integration of XML schemas: A semantic approach. In *Proc. of*

International Conference on Conceptual Modeling (ER'03), LNCS 2813, pages 520–533, 2003.

[Zam04] L. Zamboulis. XML data integration by graph restructuring. In *Proc. of British National Conference on Databases (BNCOD'04), LNCS 3112*, pages 57–71, Springer, 2004.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. of ACM SIGMOD'95*, pages 316–327, 1995.

[ZGMW96] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. The strobe algorithms for multi-source warehouse consistency. In *Proc. of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS'96)*, pages 146–157. IEEE Computer Society, 1996.

[ZGMW98] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998.

[ZP04] L. Zamboulis and A. Poulovassilis. Using AutoMed for XML data transformation and integration. In *proc. of International Workshop on Data Integration over the Web (DIWeb'04)*, pages 58–69, 2004.

Appendix A

Proof of Theorem 1

For a tracing tuple t in the view $\mathtt{v} = \mathtt{q}(\mathbb{D})$ over a sequence of bags $\mathbb{D} = \langle \mathtt{D}_1, ..., \mathtt{D}_n \rangle$, the tracing queries $\mathtt{TQ}_{\mathbb{D}}^{AP}(t)$ and $\mathtt{TQ}_{\mathbb{D}}^{OP}(t)$ in Theorem 1 satisfy Definition 1 and 2 respectively. That is, letting $\mathtt{q}_{\mathbb{D}}^{AP} = \langle \mathtt{T}_1^{ap}, ..., \mathtt{T}_n^{ap} \rangle$ and $\mathtt{q}_{\mathbb{D}}^{OP} = \langle \mathtt{T}_1^{op}, ..., \mathtt{T}_n^{op} \rangle$ denote the results of $\mathtt{TQ}_{\mathbb{D}}^{AP}(t)$ and $\mathtt{TQ}_{\mathbb{D}}^{OP}(t)$ respectively, then the following hold:

1. $\mathtt{T}_i^{ap} \subseteq \mathtt{D}_i$ and $\mathtt{T}_i^{op} \subseteq \mathtt{D}_i$, for all $1 \le i \le n$.

2. $\mathtt{q}(\mathtt{q}_{\mathbb{D}}^{AP})$ and $\mathtt{q}(\mathtt{q}_{\mathbb{D}}^{OP})$ evaluate to a bag, $\mathtt{v}|t$, consisting of all copies of t in $\mathtt{v}$[1] (this corresponds to condition (a) of Definition 1 and 2).

3. $\forall t^* \in \mathtt{T}_i^{ap}$, $\mathtt{q}(\mathtt{T}_1^{ap}, ..., \mathtt{T}_i^{ap}|t^*, ..., \mathtt{T}_n^{ap}) \ne \emptyset$; and
   $\forall t^* \in \mathtt{T}_i^{op}$, $\mathtt{q}(\mathtt{T}_1^{op}, ..., \mathtt{T}_i^{op}|t^*, ..., \mathtt{T}_n^{op}) \ne \emptyset$
   (this corresponds to condition (c) of Definition 1 and 2).

4. $$\begin{cases} (a) & \forall \langle \mathtt{T}'_1, ..., \mathtt{T}'_n \rangle \text{ satisfying 1-3, } \mathtt{T}'_i \subseteq \mathtt{T}_i^{ap} \text{ for all } 1 \le i \le n \\ & \text{(corresponding to condition (b) of Definition 1); and} \\ (b) & \forall t^* \in \mathtt{T}_i^{op}, \\ & t^* \notin (\mathtt{D}_i -\!- \mathtt{T}_i^{op}) \text{ and } \mathtt{q}(\mathtt{T}_1^{op}, ..., [x | x \leftarrow \mathtt{T}_i^{op}; x \ne t^*], ..., \mathtt{T}_n^{op}) \ne \mathtt{v}|t \\ & \text{(corresponding to conditions (b) and (d) of Definition 2).} \end{cases}$$

[1]We use $\mathtt{v}|t$ to denote all copies of t in $\mathtt{v}$.

Proof of t1:

$$\begin{array}{lll} \text{If} & \texttt{q} & = \texttt{D}_1 ++ \ldots ++ \texttt{D}_n, \\ \text{then} & \texttt{TQ}^{AP}_{\mathbb{D}}(t) & = \texttt{TQ}^{OP}_{\mathbb{D}}(t) = \langle \texttt{D}_1|t, \ldots, \texttt{D}_n|t\rangle \end{array}$$

Suppose $\texttt{T}^* = \langle \texttt{T}^*_1, ..., \texttt{T}^*_n\rangle = \langle \texttt{D}_1|t, \ldots, \texttt{D}_n|t\rangle$

1. Clearly, $\texttt{T}^*_i \subseteq \texttt{D}_i$ for all $1 \leq i \leq n$;

2. $\texttt{q}(\texttt{T}^*) = \texttt{q}(\texttt{T}^*_1, ..., \texttt{T}^*_n) = (\texttt{D}_1|t) ++ ... ++ (\texttt{D}_n|t) = \texttt{v}|t$;

3. $\forall t^* \in \texttt{T}^*_i$, $q(\texttt{T}^*_1, ..., \texttt{T}^*_i|t^*, ..., \texttt{T}^*_n) = \texttt{T}^*_1 ++ ... ++ \texttt{T}^*_i ++ ... ++ \texttt{T}^*_n$
   $= (\texttt{D}_1|t) ++ ... ++ (\texttt{D}_n|t) \neq \emptyset$, since $t \in \texttt{v}$;

4. (a) $\forall \texttt{T}^{*\prime}_i$ satisfying 1-3, if $\texttt{T}^{*\prime}_i \not\subseteq \texttt{T}^*_i$ for some i, then:
   Either there exists $t' \in \texttt{T}^{*\prime}_i$ such that $t' \neq t$,
   $\Rightarrow t' \in \texttt{q}(\texttt{T}^*_1, ..., \texttt{T}^{*\prime}_i, ..., \texttt{T}^*_n) \Rightarrow \texttt{q}(\texttt{T}^*_1, ..., \texttt{T}^{*\prime}_i, ..., \texttt{T}^*_n) \neq \texttt{v}|t$, violating 2;
   Or $(\texttt{countNum}\ t\ \texttt{T}^{*\prime}_i) > (\texttt{countNum}\ t\ \texttt{T}^*_i)$ [2], and since
   $(\texttt{countNum}\ t\ \texttt{T}^*_i) = (\texttt{countNum}\ t\ \texttt{D}_i)$,
   $\Rightarrow (\texttt{countNum}\ t\ \texttt{T}^{*\prime}_i) > (\texttt{countNum}\ t\ \texttt{D}_i)$, violating 1.
   Therefore $\texttt{T}^{*\prime}_i \subseteq \texttt{T}^*_i$ for all $1 \leq i \leq n$.

   (b)$\texttt{D}_i -- \texttt{T}^*_i = \texttt{D}_i -- \texttt{D}_i|t = [x|x \leftarrow \texttt{D}_i; x \neq t]$
   Therefore, $\forall t^* \in \texttt{T}^*_i$, $t^* \notin (\texttt{D}_i -- \texttt{T}^*_i)$.
   Also, $[x|x \leftarrow \texttt{T}^*_i; x \neq t^*] = [x|x \leftarrow \texttt{T}^*_i; x \neq t] = \emptyset$.
   Suppose $\texttt{v'} = \texttt{q}(\texttt{T}^*_1, ..., [x|x \leftarrow \texttt{T}^*_i; x \neq t^*], ..., \texttt{T}^*_n) = \texttt{q}(\texttt{T}^*_1, ..., \emptyset, ..., \texttt{T}^*_n)$,
   then $\texttt{countNum}\ t\ \texttt{v'} = (\texttt{countNum}\ t\ \texttt{v}) - (\texttt{countNum}\ t\ \texttt{T}^*_i)$
   $\Rightarrow (\texttt{countNum}\ t\ \texttt{v'}) < (\texttt{countNum}\ t\ \texttt{v})$, if $\texttt{countNum}\ t\ \texttt{T}^*_i > 0$.
   Therefore, in general, $\texttt{q}(\texttt{T}^*_1, ..., [x|x \leftarrow \texttt{T}^*_i; x \neq t^*], ..., \texttt{T}^*_n) \neq \texttt{v}|t$

Proof of t2:

$$\begin{array}{lll} \text{If} & \texttt{q} & = \texttt{D}_1 -- \texttt{D}_2 \\ \text{then} & \texttt{TQ}^{AP}_{\mathbb{D}}(t) & = \langle \texttt{D}_1|t, \texttt{D}_2\rangle \\ \text{and} & \texttt{TQ}^{OP}_{\mathbb{D}}(t) & = \langle \texttt{D}_1|t, \texttt{D}_2|t\rangle \end{array}$$

[2]Function `countNum a D` returns the number of occurrences of the data item `a` in the bag `D`, *i.e.* $\texttt{countNum a D} = \texttt{count}\ [\texttt{x}|\texttt{x} \leftarrow \texttt{D}; \texttt{x} = \texttt{a}]$.

Let $\mathtt{q}_{\mathbb{D}}^{AP} = \langle \mathtt{T}_1^{ap}, \mathtt{T}_2^{ap} \rangle = \langle \mathtt{D}_1|t, \mathtt{D}_2 \rangle$

and $\mathtt{q}_{\mathbb{D}}^{OP} = \langle \mathtt{T}_1^{op}, \mathtt{T}_2^{op} \rangle = \langle \mathtt{D}_1|t, \mathtt{D}_2|t \rangle$

1. Clearly, $\mathtt{T}_i^{ap} \subseteq \mathtt{D}_i$ and $\mathtt{T}_i^{op} \subseteq \mathtt{D}_i$ for all $1 \leq i \leq n$;

2. $\mathtt{q}(\mathtt{q}_{\mathbb{D}}^{AP}) = \mathtt{D}_1|t -- \mathtt{D}_2 = (\mathtt{D}_1 -- \mathtt{D}_2)|t = \mathtt{v}|t$

   $\mathtt{q}(\mathtt{q}_{\mathbb{D}}^{OP}) = \mathtt{D}_1|t -- \mathtt{D}_2|t = (\mathtt{D}_1 -- \mathtt{D}_2)|t = \mathtt{v}|t$;

3. $\forall t^* \in \mathtt{T}_1^{ap}$, it must be the case that $t^* = t$,

   $\Rightarrow \mathtt{T}_1^{ap}|t^* = \mathtt{T}_1^{ap}|t = \mathtt{D}_1|t$

   Therefore $\mathtt{q}(\mathtt{T}_1^{ap}|t^*, \mathtt{T}_2^{ap}) = \mathtt{D}_1|t -- \mathtt{D}_2 = (\mathtt{D}_1 -- \mathtt{D}_2)|t = \mathtt{v}|t \neq \emptyset$

   Similarly, $\forall t' \in \mathtt{T}_2^{ap}$, we have $\mathtt{q}(\mathtt{T}_1^{ap}, \mathtt{T}_2^{ap}|t') = \mathtt{D}_1|t -- \mathtt{D}_2|t' \neq \emptyset$

   For $\mathtt{q}_{\mathbb{D}}^{OP}$, the proof is similar.

4. (a) For any $\langle \mathtt{T}_1^{ap'}, \mathtt{T}_2^{ap'} \rangle$ satisfying 1-3,

   because $\mathtt{T}_1^{ap} = \mathtt{D}_1|t$ and $\mathtt{T}_1^{ap'} \subseteq \mathtt{D}_1$,

   if $\mathtt{T}_1^{ap'} \nsubseteq \mathtt{T}_1^{ap}$, then there exists $t' \in \mathtt{T}_1^{ap'}$ such that $t' \neq t$.

   Because $\mathtt{q}(\langle \mathtt{T}_1^{ap'}, \mathtt{T}_2^{ap'} \rangle) = \mathtt{T}_1^{ap'} -- \mathtt{T}_2^{ap'} = \mathtt{v}|t$,

   therefore $t' \in \mathtt{T}_2^{ap'}$ and $\mathtt{q}(\langle [t'], \mathtt{T}_2^{ap'} \rangle) = [t'] -- \mathtt{T}_2^{ap'} = \emptyset$, violating 3.

   Therefore $\mathtt{T}_1^{ap'} \subseteq \mathtt{T}_1^{ap}$.

   Because $\mathtt{T}_2^{ap} = \mathtt{D}_2$ and $\mathtt{T}_2^{ap'} \subseteq \mathtt{D}_2$, we have $\mathtt{T}_2^{ap'} \subseteq \mathtt{T}_2^{ap}$.

   (b) $\forall t^* \in \mathtt{T}_i^{op}$, we have $t^* = t$.

   Because $\mathtt{T}_i^{op} = \mathtt{D}_i|t$,

   $\mathtt{D}_i -- \mathtt{T}_i^{op} = \mathtt{D}_i -- \mathtt{D}_i|t = [x|x \leftarrow \mathtt{D}_i; x \neq t]$

   Therefore $t^* \notin (\mathtt{D}_i -- \mathtt{T}_i^{op})$.

   Also, because $t^* = t$,

   $[x|x \leftarrow \mathtt{T}_i^{op}; x \neq t^*] = [x|x \leftarrow \mathtt{T}_i^{op}; x \neq t] = \emptyset$

   Therefore $\mathtt{q}([x|x \leftarrow \mathtt{T}_1^{op}; x \neq t^*], \mathtt{T}_2^{op}) = \mathtt{q}(\emptyset, \mathtt{T}_2^{op}) = \emptyset \neq \mathtt{v}|t$

   and $\mathtt{q}(\mathtt{T}_1^{op}, [x|x \leftarrow \mathtt{T}_2^{op}; x \neq t^*]) = \mathtt{q}(\mathtt{T}_1^{op}, \emptyset) = \mathtt{T}_1^{op} = \mathtt{D}_1|t \neq \mathtt{v}|t$ in general.

Proof of t3:

$$\begin{array}{lll} \text{If} & \mathtt{q} & = \mathtt{group}\ \mathtt{D} \\ \text{then} & \mathtt{TQ}_{\mathbb{D}}^{AP}(t) & = \mathtt{TQ}_{\mathbb{D}}^{OP}(t) \ = [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t] \end{array}$$

Let $\mathtt{T}^* = \mathtt{q}_{\mathbb{D}}^{AP} = \mathtt{q}_{\mathbb{D}}^{OP} = [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t]$

1. Clearly $\mathtt{T}^* \subseteq \mathtt{D}$.

2. $\mathtt{q}(\mathtt{T}^*) = \mathtt{group}\ [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t]$

   $= [x|x \leftarrow \mathtt{group}\ \mathtt{D}; x = t] = \mathtt{v}|t$

3. $\forall t^* \in \mathtt{T}^*, \mathtt{q}(\mathtt{T}^*|t^*) = \mathtt{group}\ (\mathtt{T}^*|t^*) \neq \emptyset$

4. (a) Suppose $\mathtt{T}^{*\prime}$ satisfies 1-3.

   If $\mathtt{T}^{*\prime} \nsubseteq \mathtt{T}^*$, then there exists $t^{*\prime} \in \mathtt{T}^{*\prime}$ such that $(\mathtt{first}\ t^{*\prime}) \neq (\mathtt{first}\ t)$

   $\Rightarrow$ there exists $t' \in \mathtt{q}(\mathtt{T}^{*\prime}) = \mathtt{group}\ \mathtt{T}^{*\prime}$ such that $(\mathtt{first}\ t') \neq (\mathtt{first}\ t)$

   $\Rightarrow \mathtt{q}(\mathtt{T}^{*\prime}) \neq \mathtt{v}|t$, violating 2

   Therefore $\mathtt{T}^{*\prime} \subseteq \mathtt{T}^*$

   (b) Because $\mathtt{D} -- \mathtt{T}^* = [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x \neq \mathtt{first}\ t]$, then

   $\forall t^* \in \mathtt{T}^*, t^* \in [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t]$ and $t^* \notin (\mathtt{D} -- \mathtt{T}^*)$

   Again, because $\mathtt{q}([x|x \leftarrow \mathtt{T}^*; x \neq t^*]) = \mathtt{group}\ (\mathtt{T}^* -- \mathtt{T}^*|t^*) \neq \mathtt{group}\ \mathtt{T}^*$

   then $\mathtt{q}([x|x \leftarrow \mathtt{T}^*; x \neq t^*]) \neq \mathtt{v}|t$

Proof of t4:

$$\begin{array}{lll} \text{If:} & \mathtt{q} & = \mathtt{sort}\ \mathtt{D}/\ \mathtt{distinct}\ \mathtt{D} \\ \text{then} & \mathtt{TQ}_{\mathbb{D}}^{AP}(t) & = \mathtt{TQ}_{\mathbb{D}}^{OP}(t) \ = \mathtt{D}|t \end{array}$$

For $\mathtt{q} = \mathtt{sort}\ \mathtt{D}$:

1. $\mathtt{T}^* = \mathtt{q}_{\mathbb{D}}^{AP} = \mathtt{q}_{\mathbb{D}}^{OP} = \mathtt{D}|t \subseteq \mathtt{D}$;

2. $\mathtt{q}(\mathtt{T}^*) = \mathtt{sort}\ \mathtt{D}|t = \mathtt{v}|t$;

3. $\forall t^* \in \mathtt{T}^*, t^* = t$, and therefore

   $\mathtt{q}(\mathtt{T}^*|t^*) = \mathtt{sort}\ \mathtt{T}^*|t \neq \emptyset$;

4. (a) Suppose $\mathtt{T}^{*\prime}$ satisfies 1-3.

   If $\mathtt{T}^{*\prime} \nsubseteq \mathtt{T}^*$, then there exists $t' \in \mathtt{T}^{*\prime}$ such that $t' \neq t$

   $\Rightarrow t' \in \mathtt{q}(\mathtt{T}^{*\prime}) = \mathtt{sort}\ \mathtt{T}^{*\prime}$

   $\Rightarrow \mathtt{q}(\mathtt{T}^{*\prime}) \neq \mathtt{v}|t$, violating 2

   Therefore $\mathtt{T}^{*\prime} \subseteq \mathtt{T}^*$

   (b) Because $\mathtt{D} -- \mathtt{T}^* = [x|x \leftarrow \mathtt{D}; x \neq t]$ and $\forall t^* \in \mathtt{T}^*, t^* = t$,

   therefore $t^* \notin (\mathtt{D} -- \mathtt{T}^*)$.

   Also, because

   $\mathtt{q}([x|x \leftarrow \mathtt{T}^*; x \neq t^*]) = \mathtt{q}(\mathtt{T}^* -- \mathtt{T}^*|t^*) = \mathtt{sort}\ (\mathtt{T}^* -- \mathtt{T}^*|t^*) \neq \mathtt{sort}\ \mathtt{T}^*$

   Therefore $\mathtt{q}([x|x \leftarrow \mathtt{T}^*; x \neq t^*]) \neq \mathtt{v}|t$

The proof of $\mathtt{q} = \mathtt{distinct}\ \mathtt{D}$ is similar.

Proof of t5:

$$\begin{aligned} \text{If:}\quad & \mathtt{q} &=& \ \mathtt{max}\ \mathtt{D}\ /\ \mathtt{min}\ \mathtt{D} \\ \text{then}\quad & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) &=& \ \mathtt{D} \\ \text{and}\quad & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) &=& \ \mathtt{D}|t \end{aligned}$$

For $\mathtt{q} = \mathtt{max}\ \mathtt{D}$:

1. $\mathtt{q}^{AP}_{\mathbb{D}} = \mathtt{D} \subseteq \mathtt{D}$ and $\mathtt{q}^{OP}_{\mathbb{D}} = \mathtt{D}|t \subseteq \mathtt{D}$.

2. $\mathtt{q}(\mathtt{q}^{AP}_{\mathbb{D}}) = \mathtt{q}(\mathtt{D}) = t$

   $\mathtt{q}(\mathtt{q}^{OP}_{\mathbb{D}}) = \mathtt{q}(\mathtt{D}|t) = \mathtt{max}\ t = t$

3. $\forall t^* \in \mathtt{q}^{AP}_{\mathbb{D}}, \mathtt{q}(\mathtt{q}^{AP}_{\mathbb{D}}|t^*) = \mathtt{max}\ \mathtt{D}|t^* \neq \emptyset$

   $\forall t^* \in \mathtt{q}^{OP}_{\mathbb{D}}, \mathtt{q}(\mathtt{q}^{OP}_{\mathbb{D}}|t^*) = \mathtt{max}\ \mathtt{D}|t^* \neq \emptyset$

4. (a) Clearly, $\mathtt{q}^{AP}_{\mathbb{D}} = \mathtt{D}$ is the maximal subset of $\mathtt{D}$.

   (b) Because $\mathtt{D} -- \mathtt{q}^{OP}_{\mathbb{D}} = [x|x \leftarrow \mathtt{D}; x \neq t]$ and $\forall t^* \in \mathtt{q}^{OP}_{\mathbb{D}}, t^* = t$

   then $t^* \notin (\mathtt{D} -- \mathtt{q}^{OP}_{\mathbb{D}})$ and $\mathtt{q}([x|x \leftarrow \mathtt{q}^{OP}_{\mathbb{D}}; x \neq t]) = \mathtt{q}(\emptyset) \neq t$

The proof of $\mathtt{q} = \mathtt{min}\ \mathtt{D}$ is similar.

Proof of t6:

$$\begin{array}{lrcl} \text{If:} & \mathtt{q} & = & \mathtt{sum\ D} \\ \text{then} & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{D} \\ \text{and} & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) & = & [x|x \leftarrow \mathtt{D}; x \neq 0] \end{array}$$

1. $\mathtt{q}^{AP}_{\mathbb{D}} = \mathtt{D} \subseteq \mathtt{D}$ and $\mathtt{q}^{OP}_{\mathbb{D}} = [x|x \leftarrow \mathtt{D}; x \neq 0] \subseteq \mathtt{D}$.

2. $\mathtt{q}(\mathtt{q}^{AP}_{\mathbb{D}}) = \mathtt{q}(\mathtt{D}) = t$

   $\mathtt{q}(\mathtt{q}^{OP}_{\mathbb{D}}) = \mathtt{sum}\ [x|x \leftarrow \mathtt{D}; x \neq 0] = \mathtt{sum\ D} = t$

3. $\forall t^* \in \mathtt{q}^{AP}_{\mathbb{D}}, \mathtt{q}(\mathtt{q}^{AP}_{\mathbb{D}}|t^*) = \mathtt{sum\ D}|t^* \neq \emptyset$

   $\forall t^* \in \mathtt{q}^{OP}_{\mathbb{D}}, \mathtt{q}(\mathtt{q}^{OP}_{\mathbb{D}}|t^*) = \mathtt{sum}\ [x|x \leftarrow [x|x \leftarrow \mathtt{D}; x \neq 0]; x = t^*] \neq \emptyset$

4. (a) Clearly, $\mathtt{q}^{AP}_{\mathbb{D}} = \mathtt{D}$ is the maximal subset of $\mathtt{D}$.

   (b) Because $\mathtt{D} -- \mathtt{q}^{OP}_{\mathbb{D}} = [x|x \leftarrow \mathtt{D}; x = 0] = \mathtt{D}|0$ and $\forall t^* \in \mathtt{q}^{OP}_{\mathbb{D}}, t^* \neq 0$

   then $t^* \notin (\mathtt{D} -- \mathtt{q}^{OP}_{\mathbb{D}})$

   Also, because

   $\mathtt{q}([x|x \leftarrow \mathtt{q}^{OP}_{\mathbb{D}}; x \neq t^*]) = \mathtt{sum}\ \ [x|x \leftarrow \mathtt{q}^{OP}_{\mathbb{D}}; x \neq t^*] \neq \mathtt{sum}\ \ \mathtt{q}^{OP}_{\mathbb{D}}\ \ (t^* \neq 0)$

   then $\mathtt{q}([x|x \leftarrow \mathtt{q}^{OP}_{\mathbb{D}}; x \neq t^*]) \neq \mathtt{v}|t$

Proof of t7:

$$\begin{array}{lrcl} \text{If:} & \mathtt{q} & = & \mathtt{count\ D\ /\ avg\ D} \\ \text{then} & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) \ = \ \mathtt{D} \end{array}$$

Clearly, $\mathtt{T}^* = \mathtt{q}^{AP}_{\mathbb{D}} = \mathtt{q}^{OP}_{\mathbb{D}} \ = \ \mathtt{D}$ satisfies 1,2,3.

4. (a) $\mathtt{T}^* = \mathtt{D}$ is the maximal subset of $\mathtt{D}$

   (b) Because $\mathtt{D} -- \mathtt{T}^* = \mathtt{D} -- \mathtt{D} = \emptyset$

   then $\forall t^* \in \mathtt{T}^*, t^* \notin (\mathtt{D} -- \mathtt{T}^*)$.

   Also,

   $\mathtt{count}\ [x|x \leftarrow \mathtt{T}^*; x \neq t^*] = \mathtt{count}\ [x|x \leftarrow \mathtt{D}; x \neq t^*] \neq \mathtt{count\ D}$, and

   $\mathtt{avg}\ [x|x \leftarrow \mathtt{T}^*; x \neq t^*] = \mathtt{avg}\ [x|x \leftarrow \mathtt{D}; x \neq t^*] \neq \mathtt{avg\ D}$

   Therefore $\mathtt{q}([x|x \leftarrow \mathtt{q}^{OP}_{\mathbb{D}}; x \neq t^*] \neq \mathtt{v}$

Proof of t8:

$$\begin{array}{lrcl} \text{If:} & \mathtt{q} & = & \mathtt{gc\ max\ D} \,/\, \mathtt{gc\ min\ D} \\ \text{then} & \mathtt{TQ}_{\mathbb{D}}^{AP}(t) & = & [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t] \\ \text{and} & \mathtt{TQ}_{\mathbb{D}}^{OP}(t) & = & \mathtt{D}|t \end{array}$$

For $\mathtt{q} = \mathtt{gc\ max\ D}$:

1. $\mathtt{q}_{\mathbb{D}}^{AP} = [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t] \subseteq \mathtt{D}$ and $\mathtt{q}_{\mathbb{D}}^{OP} = \mathtt{D}|t \subseteq \mathtt{D}$

2. $\mathtt{q}(\mathtt{q}_{\mathbb{D}}^{AP}) = \mathtt{gc\ max}\ [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t] = [t]$

   $\mathtt{q}(\mathtt{q}_{\mathbb{D}}^{OP}) = \mathtt{gc\ max\ D}|t = [t]$

3. $\forall t^* \in \mathtt{q}_{\mathbb{D}}^{AP}, \mathtt{q}(\mathtt{q}_{\mathbb{D}}^{AP}|t^*) = \mathtt{gc\ max\ D}|t^* \neq \emptyset$

   $\forall t^* \in \mathtt{q}_{\mathbb{D}}^{OP}, t^* = t \Rightarrow \mathtt{q}(\mathtt{q}_{\mathbb{D}}^{OP}|t^*) = \mathtt{gc\ max}\ \mathtt{q}_{\mathbb{D}}^{OP}|t = [t] \neq \emptyset$

4. (a) Suppose $\mathtt{T}^{*\prime}$ satisfies 1-3.

   If $\mathtt{T}^{*\prime} \nsubseteq \mathtt{q}_{\mathbb{D}}^{AP}$, then there exists $t^{*\prime} \in \mathtt{T}^{*\prime}$ such that $(\mathtt{first}\ t^{*\prime}) \neq (\mathtt{first}\ t)$

   $\Rightarrow$ there exists $t' \in \mathtt{q}(\mathtt{T}^{*\prime}) = \mathtt{gc\ max}\ \mathtt{T}^{*\prime}$ such that $(\mathtt{first}\ t') = (\mathtt{first}\ t^{*\prime})$

   $\Rightarrow (\mathtt{first}\ t') \neq (\mathtt{first}\ t) \Rightarrow \mathtt{q}(\mathtt{T}^{*\prime}) \neq \mathtt{v}|t$, violating 2

   Therefore $\mathtt{T}^{*\prime} \subseteq \mathtt{q}_{\mathbb{D}}^{AP}$

   (b) Because $\mathtt{D} -- \mathtt{q}_{\mathbb{D}}^{OP} = [x|x \leftarrow \mathtt{D}; x \neq t]$ and $\forall t^* \in \mathtt{q}_{\mathbb{D}}^{OP}, t^* = t$

   then $t^* \notin (\mathtt{D} -- \mathtt{q}_{\mathbb{D}}^{OP})$

   Also, $\mathtt{q}([x|x \leftarrow \mathtt{q}_{\mathbb{D}}^{OP}; x \neq t]) = \mathtt{q}(\emptyset) \neq \mathtt{v}|t$

The proof of $\mathtt{q} = \mathtt{gc\ min\ D}$ is similar.

Proof of t9:

$$\begin{array}{lrcl} \text{If:} & \mathtt{q} & = & \mathtt{gc\ sum\ D} \\ \text{then} & \mathtt{TQ}_{\mathbb{D}}^{AP}(t) & = & [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t] \\ \text{and} & \mathtt{TQ}_{\mathbb{D}}^{OP}(t) & = & [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t; \mathtt{second}\ x \neq 0] \end{array}$$

1. $\mathtt{q}_{\mathbb{D}}^{AP} = [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t] \subseteq \mathtt{D}$;

   $\mathtt{q}_{\mathbb{D}}^{OP} = [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t; \mathtt{second}\ x \neq 0] \subseteq \mathtt{D}$.

2. $\mathtt{q}(\mathtt{q}_{\mathbb{D}}^{AP}) = \mathtt{gc\ \ sum}\ [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t] = t$

   $\mathtt{q}(\mathtt{q}_{\mathbb{D}}^{OP}) = \mathtt{gc\ \ sum}\ [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t; \mathtt{second}\ x \neq 0] = t$

3. $\forall t^* \in \mathtt{q}_{\mathbb{D}}^{AP}, \mathtt{q}(\mathtt{q}_{\mathbb{D}}^{AP}|t^*) = \mathtt{gc\ sum}\ \mathtt{q}_{\mathbb{D}}^{AP}|t^* \neq \emptyset$

   $\forall t^* \in \mathtt{q}_{\mathbb{D}}^{OP}, \mathtt{q}(\mathtt{q}_{\mathbb{D}}^{OP}|t^*) = \mathtt{gc\ \ sum}\ \mathtt{q}_{\mathbb{D}}^{OP}|t^* \neq \emptyset$

4. (a) Suppose $\mathtt{T}^{*\prime}$ satisfies 1-3.

   If $\mathtt{T}^{*\prime} \nsubseteq \mathtt{q}_{\mathbb{D}}^{AP}$, then there exists $t^{*\prime} \in \mathtt{T}^{*\prime}$ such that $(\mathtt{first}\ t^{*\prime}) \neq (\mathtt{first}\ t)$

   $\Rightarrow$ there exists $t' \in \mathtt{q}(\mathtt{T}^{*\prime}) = \mathtt{gc\ \ sum}\ \mathtt{T}^{*\prime}$ such that $(\mathtt{first}\ t') = (\mathtt{first}\ t^{*\prime})$

   $\Rightarrow (\mathtt{first}\ t') \neq (\mathtt{first}\ t) \Rightarrow \mathtt{q}(\mathtt{T}^{*\prime}) \neq \mathtt{v}|t$, violating 2

   Therefore $\mathtt{T}^{*\prime} \subseteq \mathtt{q}_{\mathbb{D}}^{AP}$

   (b) Because $\mathtt{D} -- \mathtt{q}_{\mathbb{D}}^{OP} = [x|x \leftarrow \mathtt{D}; (\mathtt{first}\ x \neq \mathtt{first}\ t)\ \mathtt{or}\ (\mathtt{second}\ x = 0)]$

   then $\mathtt{q}_{\mathbb{D}}^{OP} \nsubseteq (\mathtt{D} -- \mathtt{q}_{\mathbb{D}}^{OP}) \Rightarrow \forall t^* \in \mathtt{q}_{\mathbb{D}}^{OP}, t^* \notin (\mathtt{D} -- \mathtt{q}_{\mathbb{D}}^{OP})$

   Also, because $(\mathtt{second}\ t^*) \neq 0$

   then $\mathtt{q}([x|x \leftarrow \mathtt{q}_{\mathbb{D}}^{OP}; x \neq t^*]) = \mathtt{gc\ \ sum}([x|x \leftarrow \mathtt{q}_{\mathbb{D}}^{OP}; x \neq t^*]) \neq \mathtt{gc\ \ sum}\ \mathtt{q}_{\mathbb{D}}^{OP}$

   Therefore $\mathtt{q}([x|x \leftarrow \mathtt{q}_{\mathbb{D}}^{OP}; x \neq t^*] \neq \mathtt{v}|t$

Proof of t10:

$$\begin{array}{lrcl} \text{If:} & \mathtt{q} & = & \mathtt{gc\ \ count\ D\ /\ gc\ \ avg\ D} \\ \text{then} & \mathtt{TQ}_{\mathbb{D}}^{AP}(t) & = & \mathtt{TQ}_{\mathbb{D}}^{OP}(t) \ = [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t] \end{array}$$

The proof of $\mathtt{T}^* = [x|x \leftarrow \mathtt{D}; \mathtt{first}\ x = \mathtt{first}\ t]$ satisfying $1, 2, 3$ and 4 is similar to above `gc f` functions.

Proof of t11:

If: $\mathtt{q(D)} = [\overline{x}|\overline{x_1} \leftarrow \mathtt{D}_1; \ldots; \overline{x_n} \leftarrow \mathtt{D}_n; C_1; ...; C_k]$

then $\mathtt{TQ}^{AP}_{\mathbb{D}}(t) = \mathtt{TQ}^{OP}_{\mathbb{D}}(t) = \langle [x_1|x_1 \leftarrow \mathtt{D}_1; x_1 = (\mathtt{lambda}\ \overline{x}.\overline{x_1})\ t], \ldots,$
$[x_n|x_n \leftarrow \mathtt{D}_n; x_n = (\mathtt{lambda}\ \overline{x}.\overline{x_n})\ t]\rangle$

Suppose $\mathtt{T}^* = \langle \mathtt{T}^*_1, ..., \mathtt{T}^*_n\rangle = \langle [x_1|x_1 \leftarrow \mathtt{D}_1; x_1 = (\mathtt{lambda}\ \overline{x}.\overline{x_1})\ t], \ldots, [x_n|x_n \leftarrow \mathtt{D}_n; x_n = (\mathtt{lambda}\ \overline{x}.\overline{x_n})\ t]\rangle$

1. Clearly, $\mathtt{T}^*_i \subseteq \mathtt{D}_i$, for all $1 \leq i \leq n$;

2. Suppose $\overline{x} = \{\overline{x_1}, ..., \overline{x_n}\}$ (without loss of generality), and

   $t = \{t_1, ..., t_n\}$ where $t_i = (\mathtt{lambda}\ \overline{x}.\overline{x_i}\ t)$.

   $$\begin{aligned} \mathtt{q}(\mathtt{T}^*) &= \mathtt{q}(\mathtt{T}^*_1, ..., \mathtt{T}^*_n) = [\overline{x}|\overline{x_1} \leftarrow \mathtt{T}^*_1; \ldots; \overline{x_n} \leftarrow \mathtt{T}^*_n; C_1; ...; C_k] \\ &= [\{\overline{x_1}, \ldots, \overline{x_n}\}|\overline{x_1} \leftarrow [x|x \leftarrow \mathtt{D}_1; x = t_1]; \ldots; \\ &\qquad \overline{x_n} \leftarrow [x|x \leftarrow \mathtt{D}_n; x = t_n]; C_1; ...; C_k] \end{aligned}$$

   Because $t = \{t_1, ..., t_n\}$ satisfies predicates $C_1; ...; C_k$, then

   $\mathtt{q}(\mathtt{T}^*) = [\{\overline{x_1}, \ldots, \overline{x_n}\}|\overline{x_1} \leftarrow \mathtt{D}_1; \ldots; \overline{x_n} \leftarrow \mathtt{D}_n; \{\overline{x_1}, ..., \overline{x_n}\} = \{t_1, ..., t_n\}] = \mathtt{v}|t$

3. Because $\forall t^* \in \mathtt{T}^*_i$, $t^* = t_i$

   $\mathtt{q}(\mathtt{T}^*_1, ..., \mathtt{T}^*_i|t^*, ..., \mathtt{T}^*_n) = [\overline{x}|\overline{x_1} \leftarrow \mathtt{T}^*_1; \ldots; \overline{x_i} \leftarrow \mathtt{T}^*_i|t^*; \ldots; \overline{x_n} \leftarrow \mathtt{T}^*_n; C_1; ...; C_k]$

   Therefore $\mathtt{q}(\mathtt{T}^*_1, ..., \mathtt{T}^*_i|t^*, ..., \mathtt{T}^*_n) \neq \emptyset$;

4. (a) Suppose $\mathtt{T}^{*\prime} = \langle \mathtt{T}'_1, ..., \mathtt{T}'_n\rangle$ satisfies 1-3.

   If $\mathtt{T}^{*\prime}_i \not\subseteq \mathtt{T}^*_i$ for some i, then there exists $t^{*\prime}_i \in \mathtt{T}^{*\prime}_i$ such that $t^{*\prime}_i \neq t_i$

   $\Rightarrow \mathtt{q}(\mathtt{T}^*_1; ...; [t^{*\prime}_i]; ...; \mathtt{T}^*_n) \neq \mathtt{v}|t$

   Also, because $\mathtt{q}(\mathtt{T}^*_1; ...; [t^{*\prime}_i]; ...; \mathtt{T}^*_n) \subseteq \mathtt{q}(\mathtt{T}^*)$, and $\mathtt{q}(\mathtt{T}^*) = \mathtt{v}|t$

   $\Rightarrow \mathtt{q}(\mathtt{T}^*_1; ...; [t^{*\prime}_i]; ...; \mathtt{T}^*_n) = \emptyset$, violating 3

   Therefore $\mathtt{T}^{*\prime} \subseteq \mathtt{T}^*$

   (b) Because $\mathtt{T}^*_i = \mathtt{D}_i|t_i$, then $\forall t^* \in \mathtt{T}^*_i$, $t^* = t_i$ and

   $\mathtt{D}_i -- \mathtt{T}^*_i = \mathtt{D}_i -- \mathtt{D}_i|t_i = [x|x \leftarrow \mathtt{D}_i; x \neq t_i]$

   Therefore $t^* \notin (\mathtt{D}_i -- \mathtt{T}^*_i)$

   Also, $[x|x \leftarrow \mathtt{T}^*_i; x \neq t^*] = [x|x \leftarrow \mathtt{T}^*_i; x \neq t_i] = \emptyset$, therefore

$\mathtt{q}(\mathtt{T}_1^*, ..., [x|x \leftarrow \mathtt{T}_i^*; x \neq t^*], ..., \mathtt{T}_n^*) = \mathtt{q}(\mathtt{T}_1^*, ..., \emptyset, ..., \mathtt{T}_n^*)$

$= [\overline{x}|\overline{x_1} \leftarrow \mathtt{T}_1^*; ...; \overline{x_i} \leftarrow \emptyset; ...; \overline{x_n} \leftarrow \mathtt{T}_n^*; C_1; ...; C_k]$

$= \emptyset \neq \mathtt{v}|t$

Proof of t12:

If: $\mathtt{q} \quad = \quad [\overline{x}|\overline{x} \leftarrow \mathtt{D}_1; \mathtt{member}\ \mathtt{D}_2\ \overline{y}]$

then $\mathtt{TQ}_{\mathbb{D}}^{AP}(t) \quad = \quad \mathtt{TQ}_{\mathbb{D}}^{OP}(t) \quad = \langle \mathtt{D}_1|t, [y|y \leftarrow \mathtt{D}_2; y = (\mathtt{lambda}\ \overline{x}.\overline{y})\ t]\rangle$

Suppose $\overline{x} = \overline{y}$ (without loss of generality), and let $\langle \mathtt{T}_1^*, \mathtt{T}_2^* \rangle = \langle \mathtt{D}_1|t, \mathtt{D}_2|t \rangle$

1. Clearly, $\mathtt{T}_1^* \subseteq \mathtt{D}_1$ and $\mathtt{T}_2^* \subseteq \mathtt{D}_2$.

2. $\mathtt{q}(\langle \mathtt{T}_1^*, \mathtt{T}_2^* \rangle) = [x|x \leftarrow \mathtt{D}_1|t; \mathtt{member}\ \mathtt{D}_2|t\ x] = \mathtt{v}|t$

3. $\forall t^* \in \mathtt{T}_i^*, t^* = t \Rightarrow \mathtt{T}_i^*|t^* = \mathtt{T}_i^*|t = \mathtt{T}_i^*$

   Therefore $\mathtt{q}(\langle \mathtt{T}_1^*|t^*, \mathtt{T}_2^*|t^* \rangle) \neq \emptyset$

4. (a) Suppose $\langle \mathtt{T}'_1, \mathtt{T}'_2 \rangle$ satisfies 1-3.

   If $\mathtt{T}'_i \not\subseteq \mathtt{T}_i^*$, then there exists $t' \in \mathtt{T}'_i$ such that $t' \neq t$.

   If $t' \in \mathtt{T}'_1$ and $t' \in \mathtt{T}'_2$, then $t' \in \mathtt{q}(\langle \mathtt{T}'_1, \mathtt{T}'_2 \rangle) \Rightarrow \mathtt{q}(\langle \mathtt{T}'_1, \mathtt{T}'_2 \rangle) \neq \mathtt{v}|t$, violating 2;

   else if $t' \in \mathtt{T}'_1$ and $t' \notin \mathtt{T}'_2$, then $\mathtt{q}([t'], \mathtt{T}'_2) = \emptyset$, violating 3;

   else if $t' \notin \mathtt{T}'_1$ and $t' \in \mathtt{T}'_2$, then $\mathtt{q}(\mathtt{T}'_1, [t']) = \emptyset$, violating 3.

   Therefore $\mathtt{T}'_i \subseteq \mathtt{T}_i^*$

   (b)Because $\forall t^* \in \mathtt{T}_i^*, t^* = t$, then

   $\mathtt{D}_i -- \mathtt{T}_i^* = [x|x \leftarrow \mathtt{D}_i; x \neq t]$ and

   $t^* \notin (\mathtt{D}_i -- \mathtt{T}_i^*)$

   Also, because $[x|x \leftarrow \mathtt{T}_i^*; x \neq t^*] = \emptyset$

   then $\mathtt{q}(\mathtt{T}_1^*, [x|x \leftarrow \mathtt{T}_2^*; x \neq t^*]) = \mathtt{q}([x|x \leftarrow \mathtt{T}_1^*; x \neq t^*], \mathtt{T}_2^*) = \emptyset \neq \mathtt{v}|t$

Proof of t13:

$$\begin{array}{lrcl} \text{If:} & \mathtt{q} & = & [\overline{x}|\overline{x} \leftarrow \mathtt{D}_1; \mathtt{not}(\mathtt{member}\ \mathtt{D}_2\ \overline{y})] \\ \text{then} & \mathtt{TQ}^{AP}_{\mathbb{D}}(t) & = & \langle \mathtt{D}_1|t, \mathtt{D}_2\rangle \\ \text{and} & \mathtt{TQ}^{OP}_{\mathbb{D}}(t) & = & \langle \mathtt{D}_1|t, \emptyset\rangle \end{array}$$

Let $\mathtt{q}^{AP}_{\mathbb{D}} = \langle \mathtt{T}^{ap}_1, \mathtt{T}^{ap}_2\rangle = \langle \mathtt{D}_1|t, \mathtt{D}_2\rangle$

and $\mathtt{q}^{OP}_{\mathbb{D}} = \langle \mathtt{T}^{op}_1, \mathtt{T}^{op}_2\rangle = \langle \mathtt{D}_1|t, \emptyset\rangle$

1. Clearly, $\mathtt{T}^{ap}_i \subseteq \mathtt{D}_i$ and $\mathtt{T}^{op}_i \subseteq \mathtt{D}_i$, for i = 1,2;

2. $\mathtt{q}(\mathtt{q}^{AP}_{\mathbb{D}}) = [x|x \leftarrow \mathtt{D}_1|t; \mathtt{not}\ (\mathtt{member}\ \mathtt{D}_2\ x)] = \mathtt{v}|t$

   $\mathtt{q}(\mathtt{q}^{OP}_{\mathbb{D}}) = [x|x \leftarrow \mathtt{D}_1|t; \mathtt{not}\ (\mathtt{member}\ \emptyset\ x)] = \mathtt{v}|t$

3. $\forall t^*_1 \in \mathtt{T}^{ap}_1, t^*_1 = t$ and $\mathtt{T}^{ap}_1|t^*_1 = \mathtt{T}^{ap}_1|t = \mathtt{T}^{ap}_1$

   Therefore $\mathtt{q}(\mathtt{T}^{ap}_1|t^*_1, \mathtt{T}^{ap}_2) = \mathtt{q}(\mathtt{T}^{ap}_1, \mathtt{T}^{ap}_2) = \mathtt{q}(\mathtt{q}^{ap}_{\mathbb{D}}) = \mathtt{v}|t \neq \emptyset$

   Because $t \notin \mathtt{D}_2 \Rightarrow \forall t^*_2 \in \mathtt{T}^{ap}_2, t^*_2 \neq t$

   then $\mathtt{q}(\mathtt{T}^{ap}_1, \mathtt{T}^{ap}_2|t^*_2) = [x|x \leftarrow \mathtt{D}_1|t; \mathtt{not}\ (\mathtt{member}\ \mathtt{D}_2|t^*_2\ x)] = \mathtt{D}_1|t \neq \emptyset$

   For $\mathtt{q}^{OP}_{\mathbb{D}}$, the proof is similar.

4. (a) Suppose $\langle \mathtt{T}'_1, \mathtt{T}'_2\rangle$ satisfies 1-3.

   If $\mathtt{T}'_1 \nsubseteq \mathtt{T}^{AP}_1$, then there exists $t'_1 \in \mathtt{T}'_1$ such that $t'_1 \neq t$.

   If $t'_1 \notin \mathtt{T}'_2$ then $t'_1 \in \mathtt{q}(\langle \mathtt{T}'_1, \mathtt{T}'_2\rangle) \Rightarrow \mathtt{q}(\langle \mathtt{T}'_1, \mathtt{T}'_2\rangle) \neq \mathtt{v}|t$, violating 2;

   else if $t'_1 \in \mathtt{T}'_2$, then $\mathtt{q}([t'_1], \mathtt{T}'_2) = \emptyset$, violating 3

   Therefore $\mathtt{T}'_1 \subseteq \mathtt{T}^{ap}_1$

   Because $\mathtt{T}^{ap}_2 = \mathtt{D}_2$ then $\mathtt{T}'_2 \subseteq \mathtt{T}^{ap}_2$.

   (b)$\forall t^*_1 \in \mathtt{T}^{op}_1, t^*_1 = t$ and $(\mathtt{D}_1 -- \mathtt{T}^{op}_1) = [x|x \leftarrow \mathtt{D}_1; x \neq t]$

   Therefore $t^*_1 \notin (\mathtt{D}_1 -- \mathtt{T}^{op}_1)$

   Also, because $[x|x \leftarrow \mathtt{T}^{op}_1; x \neq t^*_1] = [x|x \leftarrow \mathtt{T}^{op}_1; x \neq t] = \emptyset$

   Therefore $\mathtt{q}([x|x \leftarrow \mathtt{T}^{op}_1; x \neq t^*], \mathtt{T}^{op}_2) = \emptyset \neq \mathtt{v}|t$

   There is no need to consider $\mathtt{T}^{OP}_2$ since it is $\emptyset$ by definition.

Proof of t14:

If: $\texttt{q} = \texttt{map (lambda p1.p2) D}$

then $\texttt{TQ}^{AP}_{\mathbb{D}}(t) = \texttt{TQ}^{OP}_{\mathbb{D}}(t) = [p_1|p_1 \leftarrow \texttt{D}; p_2 = t]$

Let $\texttt{T}^* = \texttt{q}^{AP}_{\mathbb{D}} = \texttt{q}^{OP}_{\mathbb{D}} = [p_1|p_1 \leftarrow \texttt{D}; p_2 = t]$

1. Clearly, $\texttt{T}^* \subseteq \texttt{D}$.

2. $\texttt{q}(\texttt{T}^*) = \texttt{map (lambda } p_1.p_2) \ [p_1|p_1 \leftarrow \texttt{D}; p_2 = t] = \texttt{v}|t$

3. $\forall t^* \in \texttt{T}^*, ((\texttt{lambda } p_1.p_2) \ t^*) = t$

   Therefore $\texttt{q}(\texttt{T}^*|t^*) = \texttt{map (lambda } p_1.p_2) \ [p_1|p_1 \leftarrow \texttt{T}^*|t^*; p_2 = t] = t^* \neq \emptyset$

4. (a) Suppose $\texttt{T}'$ satisfies 1-3.

   If $\texttt{T}' \not\subseteq \texttt{T}^*$, then there exists $t' \in \texttt{T}'$ such that $((\texttt{lambda } p_1.p_2) \ t') \neq t$

   $\Rightarrow \texttt{q}(\texttt{T}'|t') = \texttt{map (lambda } p_1.p_2) \ [p_1|p_1 \leftarrow \texttt{T}'|t'; p_2 = t] = \emptyset$, violating 3

   Therefore $\texttt{T}' \subseteq \texttt{T}^*$

   (b) For any $t^* \in \texttt{T}^*$,

   because $(\texttt{D} -- \texttt{T}^*) = [p_1|p_1 \leftarrow \texttt{D}; p_1 \neq t^*] = [p_1|p_1 \leftarrow \texttt{D}; p_2 \neq t]$

   then $t^* \notin (\texttt{D} -- \texttt{T}^*)$

   Also,

   because $[p_1|p_1 \leftarrow \texttt{T}^*; p_1 \neq t^*] = [p_1|p_1 \leftarrow \texttt{T}^*; ((\texttt{lambda } p_1.p_2) \ t^*) \neq t] = \emptyset$

   Therefore $\texttt{q}([x|x \leftarrow \texttt{T}^*; x \neq t^*]) = \emptyset \neq \texttt{v}|t$

Appendix B

Justifications of IVM Formulae

B.1 Justification of IVM Formulae for $\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}$

Suppose that $\mathtt{v} = \mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}$. Then

$$\begin{aligned}
\mathtt{v}^{\mathtt{new}} &= \mathtt{D1}^{\mathtt{new}} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} \\
&= (\mathtt{D1}{+}{+}\ \triangle\mathtt{D1} -- \triangledown\mathtt{D1}) \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} \\
&= \mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} {+}{+}\ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} -- \triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} \\
&= \mathtt{D1} \bowtie_{\mathtt{c}} (\mathtt{D2}{+}{+}\ \triangle\mathtt{D2} -- \triangledown\mathtt{D2}) {+}{+}\ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} -- \triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} \\
&= (\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2} {+}{+} \mathtt{D1} \bowtie_{\mathtt{c}} \triangle\mathtt{D2} -- \mathtt{D1} \bowtie_{\mathtt{c}} \triangledown\mathtt{D2}) {+}{+}\ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} -- \triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} \\
&= (\mathtt{v} {+}{+} \mathtt{D1} \bowtie_{\mathtt{c}} \triangle\mathtt{D2} -- \mathtt{D1} \bowtie_{\mathtt{c}} \triangledown\mathtt{D2}) {+}{+}\ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} -- \triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}}
\end{aligned}$$

Because $(\mathtt{D1} \bowtie_{\mathtt{c}} \triangledown\mathtt{D2}) \subseteq \mathtt{v}$,

$$\begin{aligned}
\mathtt{v}^{\mathtt{new}} &= (\mathtt{v} {+}{+} \mathtt{D1} \bowtie_{\mathtt{c}} \triangle\mathtt{D2} {+}{+}\ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}}) -- \triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} -- \mathtt{D1} \bowtie_{\mathtt{c}} \triangledown\mathtt{D2} \\
&= (\mathtt{v} {+}{+} (\mathtt{D1}^{\mathtt{new}} {+}{+}\ \triangledown\mathtt{D1} --\ \triangle\mathtt{D1}) \bowtie_{\mathtt{c}} \triangle\mathtt{D2} {+}{+}\ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}}) \\
&\quad --\triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} -- (\mathtt{D1}^{\mathtt{new}} {+}{+}\ \triangledown\mathtt{D1} --\ \triangle\mathtt{D1}) \bowtie_{\mathtt{c}} \triangledown\mathtt{D2}) \\
&= (\mathtt{v} {+}{+} (\mathtt{D1}^{\mathtt{new}} \bowtie_{\mathtt{c}} \triangle\mathtt{D2} --\ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \triangle\mathtt{D2}) {+}{+}\ \triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \triangle\mathtt{D2} {+}{+}\ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}}) \\
&\quad --(\triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} {+}{+} (\mathtt{D1}^{\mathtt{new}} \bowtie_{\mathtt{c}} \triangledown\mathtt{D2} --\ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \triangledown\mathtt{D2}) {+}{+}\ \triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \triangledown\mathtt{D2})
\end{aligned}$$

Because $(\triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \triangle\mathtt{D2}) \subseteq (\triangledown\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}})$,

$$\begin{aligned}
\mathtt{v}^{\mathtt{new}} &= (\mathtt{v} +\!+ (\mathtt{D1}^{\mathtt{new}} \bowtie_{\mathtt{c}} \triangle\mathtt{D2} -\!- \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \triangle\mathtt{D2}) +\!+ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}}) -\!- \\
&\quad ((\nabla\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} -\!- \nabla\mathtt{D1} \bowtie_{\mathtt{c}} \triangle\mathtt{D2}) +\!+ (\mathtt{D1}^{\mathtt{new}} \bowtie_{\mathtt{c}} \nabla\mathtt{D2} -\!- \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \nabla\mathtt{D2}) \\
&\quad +\!+ \nabla\mathtt{D1} \bowtie_{\mathtt{c}} \nabla\mathtt{D2})
\end{aligned}$$

Therefore,

$$\begin{aligned}
\triangle\mathtt{v} &= (\mathtt{D1}^{\mathtt{new}} \bowtie_{\mathtt{c}} \triangle\mathtt{D2} -\!- \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \triangle\mathtt{D2}) +\!+ \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} \\
\nabla\mathtt{v} &= (\nabla\mathtt{D1} \bowtie_{\mathtt{c}} \mathtt{D2}^{\mathtt{new}} -\!- \nabla\mathtt{D1} \bowtie_{\mathtt{c}} \triangle\mathtt{D2}) +\!+ (\mathtt{D1}^{\mathtt{new}} \bowtie_{\mathtt{c}} \nabla\mathtt{D2} -\!- \triangle\mathtt{D1} \bowtie_{\mathtt{c}} \nabla\mathtt{D2}) \\
&\quad +\!+ \nabla\mathtt{D1} \bowtie_{\mathtt{c}} \nabla\mathtt{D2}
\end{aligned}$$

B.2 Justification of IVM Formulae for $\mathtt{D1} \wedge \mathtt{D2}$

Suppose that $\mathtt{v}$, $\mathtt{r1}$ and $\mathtt{r2}$ are defined as follows:

$$\begin{aligned}
\mathtt{v} &= \mathtt{D1} \wedge \mathtt{D2} \\
\mathtt{r1} &= [x | x \leftarrow \triangle\mathtt{D2}; (\mathtt{countNum}\ x\ \ \triangle\mathtt{D2}) = (\mathtt{countNum}\ x\ \mathtt{D2}^{new})] \\
\mathtt{r2} &= \nabla\mathtt{D2}\ \overline{\wedge}\ \mathtt{D2}^{new}
\end{aligned}$$

The following equivalences hold for the $\wedge$ operator since the data items of $\mathtt{r1}$ are from $\triangle\mathtt{D2}$ and do not appear in $\mathtt{D2}$, and the data items of $\mathtt{r2}$ are from $\nabla\mathtt{D2}$ and do not appear in $\mathtt{D2}$ after the deletion:

$$\begin{aligned}
\mathtt{D1} \wedge (\mathtt{D2} +\!+ \mathtt{r1} -\!- \mathtt{r2}) &= \mathtt{D1} \wedge \mathtt{D2} +\!+ \mathtt{D1} \wedge \mathtt{r1} -\!- \mathtt{D1} \wedge \mathtt{r2} \\
(\mathtt{D1} +\!+ \triangle\mathtt{D1} -\!- \nabla\mathtt{D1}) \wedge \mathtt{D2} &= \mathtt{D1} \wedge \mathtt{D2} +\!+ \triangle\mathtt{D1} \wedge \mathtt{D2} -\!- \nabla\mathtt{D1} \wedge \mathtt{D2}
\end{aligned}$$

Then,

$$\begin{aligned}
\mathtt{v}^{\mathtt{new}} &= \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{D2}^{\mathtt{new}} \\
&= \mathtt{D1}^{\mathtt{new}} \wedge (\mathtt{D2} +\!+ \mathtt{r1} -\!- \mathtt{r2}) \\
&= \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{D2} +\!+ \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} -\!- \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} \\
&= (\mathtt{D1} \wedge \mathtt{D2} +\!+ \triangle\mathtt{D1} \wedge \mathtt{D2} -\!- \nabla\mathtt{D1} \wedge \mathtt{D2}) +\!+ \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} -\!- \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} \\
&= (\mathtt{v} +\!+ \triangle\mathtt{D1} \wedge \mathtt{D2} -\!- \nabla\mathtt{D1} \wedge \mathtt{D2}) +\!+ \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} -\!- \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2}
\end{aligned}$$

Because $(\nabla\mathtt{D1} \wedge \mathtt{D2}) \subseteq \mathtt{v}$,

$$\begin{aligned}
\mathtt{v}^{\mathtt{new}} &= (\mathtt{v}\mathbin{++} \triangle\mathtt{D1} \wedge \mathtt{D2} \mathbin{++} \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1}) \mathbin{--} \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} \mathbin{--} \triangledown\mathtt{D1} \wedge \mathtt{D2}\\
&= (\mathtt{v}\mathbin{++} \triangle\mathtt{D1} \wedge \mathtt{D2} \mathbin{++} \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1}) \mathbin{--} (\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} \mathbin{++} \triangledown\mathtt{D1} \wedge \mathtt{D2})\\
&= (\mathtt{v}\mathbin{++} \triangle\mathtt{D1} \wedge (\mathtt{D2}^{\mathtt{new}} \mathbin{--} \mathtt{r1} \mathbin{++} \mathtt{r2}) \mathbin{++} \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1})\\
&\quad \mathbin{--}(\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} \mathbin{++} \triangledown\mathtt{D1} \wedge (\mathtt{D2}^{\mathtt{new}} \mathbin{--} \mathtt{r1} \mathbin{++} \mathtt{r2}))\\
&= (\mathtt{v} \mathbin{++} (\triangle\mathtt{D1} \wedge \mathtt{D2}^{\mathtt{new}} \mathbin{--} \triangle\mathtt{D1} \wedge \mathtt{r1} \mathbin{++} \triangle\mathtt{D1} \wedge \mathtt{r2}) \mathbin{++} \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1})\\
&\quad \mathbin{--}(\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} \mathbin{++} (\triangledown\mathtt{D1} \wedge \mathtt{D2}^{\mathtt{new}} \mathbin{--} \triangledown\mathtt{D1} \wedge \mathtt{r1} \mathbin{++} \triangledown\mathtt{D1} \wedge \mathtt{r2}))
\end{aligned}$$

Because $(\triangle\mathtt{D1} \wedge \mathtt{r2}) \subseteq (\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2})$,

$$\begin{aligned}
\mathtt{v}^{\mathtt{new}} &= \mathtt{v} \mathbin{++} ((\triangle\mathtt{D1} \wedge \mathtt{D2}^{\mathtt{new}} \mathbin{--} \triangle\mathtt{D1} \wedge \mathtt{r1}) \mathbin{++} \mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1}) \mathbin{--}\\
&\quad ((\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} \mathbin{--} \triangle\mathtt{D1} \wedge \mathtt{r2}) \mathbin{++} (\triangledown\mathtt{D1} \wedge \mathtt{D2}^{\mathtt{new}} \mathbin{--} \triangledown\mathtt{D1} \wedge \mathtt{r1}) \mathbin{++} \triangledown\mathtt{D1} \wedge \mathtt{r2})
\end{aligned}$$

Therefore,

$$\begin{aligned}
\triangle\mathtt{v} &= (\triangle\mathtt{D1} \wedge \mathtt{D2}^{new} \mathbin{--} \triangle\mathtt{D1} \wedge \mathtt{r1}) \mathbin{++} \mathtt{D1}^{new} \wedge \mathtt{r1}\\
\triangledown\mathtt{v} &= (\mathtt{D1}^{new} \wedge \mathtt{r2} \mathbin{--} \triangle\mathtt{D1} \wedge \mathtt{r2}) \mathbin{++} (\triangledown\mathtt{D1} \wedge \mathtt{D2}^{new} \mathbin{--} \triangledown\mathtt{D1} \wedge \mathtt{r1}) \mathbin{++} \triangledown\mathtt{D1} \wedge \mathtt{r2}
\end{aligned}$$

B.3 Justification of IVM Formulae for $\mathtt{D1} \overline{\wedge} \mathtt{D2}$

Suppose that `v`, `r1` and `r2` are defined as follows:

$$\begin{aligned}
\mathtt{v} &= \mathtt{D1} \overline{\wedge} \mathtt{D2}\\
\mathtt{r1} &= [x | x \leftarrow \triangle\mathtt{D2}; (\mathtt{countNum}\ x\ \ \triangle\mathtt{D2}) = (\mathtt{countNum}\ x\ \mathtt{D2}^{\mathtt{new}})]\\
\mathtt{r2} &= \triangledown\mathtt{D2} \overline{\wedge} \mathtt{D2}^{\mathtt{new}}
\end{aligned}$$

The following equivalences hold for the $\overline{\wedge}$ operator since the data items of `r1` are from $\triangle\mathtt{D2}$ and do not appear in `D2`, and the data items of `r2` are from $\triangledown\mathtt{D2}$ and do not appear in `D2` after the deletion:

$$\begin{aligned}
\mathtt{D1} \overline{\wedge} (\mathtt{D2} \mathbin{++} \mathtt{r1} \mathbin{--} \mathtt{r2}) &= \mathtt{D1} \overline{\wedge} \mathtt{D2} \mathbin{++} \mathtt{D1} \wedge \mathtt{r2} \mathbin{--} \mathtt{D1} \wedge \mathtt{r1}\\
(\mathtt{D1} \mathbin{++} \triangle\mathtt{D1} \mathbin{--} \triangledown\mathtt{D1}) \overline{\wedge} \mathtt{D2} &= \mathtt{D1} \overline{\wedge} \mathtt{D2} \mathbin{++} \triangle\mathtt{D1} \overline{\wedge} \mathtt{D2} \mathbin{--} \triangledown\mathtt{D1} \overline{\wedge} \mathtt{D2}
\end{aligned}$$

Then,

$$\begin{aligned}
\mathtt{v}^{\mathtt{new}} &= \mathtt{D1}^{\mathtt{new}} \,\overline{\barwedge}\, \mathtt{D2}^{\mathtt{new}} \\
&= \mathtt{D1}^{\mathtt{new}} \,\overline{\barwedge}\, (\mathtt{D2} +\!+ \,\mathtt{r1} -\!- \,\mathtt{r2}) \\
&= \mathtt{D1}^{\mathtt{new}} \,\overline{\barwedge}\, \mathtt{D2} +\!+ \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} -\!- \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} \\
&= (\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2} +\!+ \,\triangle\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2} -\!- \,\triangledown\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2}) +\!+ \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} -\!- \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} \\
&= (\mathtt{v} +\!+ \,\triangle\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2} -\!- \,\triangledown\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2}) +\!+ \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} -\!- \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1}
\end{aligned}$$

Because $(\triangledown\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2}) \subseteq \mathtt{v}$,

$$\begin{aligned}
\mathtt{v}^{\mathtt{new}} &= (\mathtt{v} +\!+ \,\triangle\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2} +\!+ \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2}) -\!- \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} -\!- \,\triangledown\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2} \\
&= (\mathtt{v} +\!+ \,\triangle\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2} +\!+ \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2}) -\!- \,(\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} +\!+ \,\triangledown\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2}) \\
&= (\mathtt{v} +\!+ \,\triangle\mathtt{D1} \,\overline{\barwedge}\, (\mathtt{D2}^{\mathtt{new}} -\!- \,\mathtt{r1} +\!+ \,\mathtt{r2}) +\!+ \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2}) \\
&\quad -\!-(\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} +\!+ \,\triangledown\mathtt{D1} \wedge (\mathtt{D2}^{\mathtt{new}} -\!- \,\mathtt{r1} +\!+ \,\mathtt{r2})) \\
&= (\mathtt{v} +\!+ \,(\triangle\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2}^{\mathtt{new}} -\!- \,\triangle\mathtt{D1} \wedge \mathtt{r2} +\!+ \,\triangle\mathtt{D1} \wedge \mathtt{r1}) +\!+ \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2}) \\
&\quad -\!-(\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} +\!+ \,(\triangledown\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2}^{\mathtt{new}} -\!- \,\triangledown\mathtt{D1} \wedge \mathtt{r2} +\!+ \,\triangledown\mathtt{D1} \wedge \mathtt{r1}))
\end{aligned}$$

Because $(\triangle\mathtt{D1} \wedge \mathtt{r1}) \subseteq (\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1})$,

$$\begin{aligned}
\mathtt{v}^{\mathtt{new}} &= \mathtt{v} +\!+ \,((\triangle\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2}^{\mathtt{new}} -\!- \,\triangle\mathtt{D1} \wedge \mathtt{r2}) +\!+ \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2}) -\!- \\
&\quad ((\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} -\!- \,\triangle\mathtt{D1} \wedge \mathtt{r1}) +\!+ \,(\triangledown\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2}^{\mathtt{new}} -\!- \,\triangledown\mathtt{D1} \wedge \mathtt{r2}) +\!+ \,\triangledown\mathtt{D1} \wedge \mathtt{r1})
\end{aligned}$$

Therefore,

$$\begin{aligned}
\triangle\mathtt{v} &= (\triangle\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2}^{\mathtt{new}} -\!- \,\triangle\mathtt{D1} \wedge \mathtt{r2}) +\!+ \,\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r2} \\
\triangledown\mathtt{v} &= (\mathtt{D1}^{\mathtt{new}} \wedge \mathtt{r1} -\!- \,\triangle\mathtt{D1} \wedge \mathtt{r1}) +\!+ \,(\triangledown\mathtt{D1} \,\overline{\barwedge}\, \mathtt{D2}^{\mathtt{new}} -\!- \,\triangledown\mathtt{D1} \wedge \mathtt{r2}) +\!+ \,\triangledown\mathtt{D1} \wedge \mathtt{r1}
\end{aligned}$$

Appendix C

Implementation of Data Warehousing Packages and API for the AutoMed Toolkit

This appendix describes the data warehousing packages and API for the AutoMed toolkit. In particular, it is the implementation of the generalised DLT algorithm described in Chapter 6. The packages and API use java and the AutoMed Repository API as the basic programming toolkits. Section C.1 discusses the structure of the data warehousing packages, Section C.2 gives a GUI supporting our DLT process, and Section C.3 gives a summary of this appendix.

C.1 Package Structure

Currently, there are three packages available in the data warehousing toolkit: dataWarehousing.dlt, dataWarehousing.util and dataWarehousing.DWExample. All packages have the prefixed hierarchy "uk.ac.bbk.automed".

C.1.1 Package uk.ac.bbk.automed.dataWarehousing.DWExample

This package gives an example of creating the AutoMed metadata for a data warehouse, *i.e.* creating the schemas of the data warehouse and AutoMed transformation pathways expressing mappings between the schemas. As described in Section 3.3, there are four steps to create the AutoMed metadata: creating AutoMed repositories, specifying data models, extracting data source schemas, and defining transformation pathways. The following three classes are used to perform these steps.

Class DefineRepository

This class is provided by the AutoMed API, which uses JDBC to access an underlying relational database and defines schemas of the repositories storing AutoMed metadata. We recall from Chapter 2 that the AutoMed repositories can be implemented using any DBMS supporting JDBC. If the DBMS of the data warehouse supports JDBC, then the AutoMed repositories can be part of the data warehouse itself.

In order to specify the URL of the DBMS and define the schema of the repositories, there are two associated config files, "`data_source_repository.cfg`" and "`reps_schema.cfg`", located in an assigned folder.

Class DefineSchemas

The class DefineSchemas has two functionalities, specifying the data models used for expressing the schemas of the data warehouse, and extracting schemas from the data sources.

Different wrapper objects are created for different kinds of data sources, for example an `OracleWrapper` is created for Oracle databases and a `PostgresWrapper` for PostgreSQL databases. The following code shows how a `PostgresWrapper` object is created:

```
PostgresWrapperFactory pwf = new PostgresWrapperFactory();
PostgresWrapper pw = (PostgresWrapper)PostgresWrapper.newAutoMedWrapper
                        (username,password,"org.postgresql.Driver",
                         "jdbc:postgresql://dbURL:5432/dbName",
                         source_schema_name,pwf);
```

Here, `username` and `password` give the username and password for accessing the PostgreSQL database; `"jdbc:postgresql://dbURL:5432/dbName"` specifies the database URL and name, and `source_schema_name` is the name of the AutoMed schema extracted from the database, which is nominated by the programmer.

Note that, `source_schema_name` given above is the name of the *source-level schema* of the database. The AutoMed toolkit defines two levels of schemas for relational databases: *source-level schemas* and *AutoMed-level schemas*. Source-level schemas are derived directly from relational databases and are used by the DBMS wrappers to query the data source data. AutoMed-level schemas are the relational schemas as described in Chapter 3. They are automatically derived from the source-level schemas by the AutoMed wrappers, and can be used by data warehouse builders as the staring point for transformation pathways. All algorithms described in this thesis are based on AutoMed-level schemas.

For example, suppose a relational database contains a table `csmarks(`<u>`sid`</u>`, sname,mark)`. The source-level schema of the database contains a construct $\langle\langle \mathsf{csmarks}, 3, \mathsf{sid}, \mathsf{sname}, \mathsf{mark}\rangle\rangle$, while the AutoMed-level schema includes the constructs $\langle\langle \mathsf{csmarks}\rangle\rangle$, $\langle\langle \mathsf{csmarks}, \mathsf{sid}\rangle\rangle$, $\langle\langle \mathsf{csmarks}, \mathsf{sname}\rangle\rangle$ and $\langle\langle \mathsf{csmarks}, \mathsf{mark}\rangle\rangle$.

The created `PostgreSQLWrapper` object `pw` can then be used to extract the schemas of the PostgreSQL database. In particular, the code

```
pw.getSchema();
```

is used to obtain the source-level schema, named by `source_schema_name`, and the code

```
pw.newAutoMedSchema(automed_schema_name);
```

is used to create the AutoMed-level schema, named by `automed_schema_name`.

Class DefineTransformations

AutoMed transformation pathways are created over the AutoMed-level schemas of the data sources. The class DefineTransformations is used to define the transformation pathway from the AutoMed-level schemas of the data sources to the AutoMed-level schema of the global database.

Suppose that `Schema` object `s` is the source schema. The code given below is used to implement the following transformations on `s`:

```
addRel (<<dept>>,           ['comp','math']);
addAtt (<<dept,d_name>>,    [{x,x} | x <- <<dept>>]);
addAtt (<<dept,avgSalary>>,[{'comp',avg[s|{n,s}<-<<comp,salary>>]},
                             {'math',avg[s|{n,s}<-<<math,salary>>]}]);
```

We firstly create a `Model` object `sql_2` specifying the relational data model supporting the SQL-2 query language, and two `Construct` objects `table` and `column` specifying the table and column constructs of this data model. Then, the method `applyAddTransformation` is used to add instances of `table` and `column` to the schema `s`:

```
Model sql_2 = Model.getModel("sql_2");
Construct table = sql_2.getConstruct("table");
Construct column = sql_2.getConstruct("column");
Schema cs = s.applyAddTransformation(table, new Object[] {"dept"},
                                                  "['comp','math']");
SchemaObject dept= cs.getSchemaObject("<<dept>>");
Schema ts = cs.applyAddTransformation(column,
                                      new Object[] {dept,"d_name"},
                                      "[{x,x} | x <- <<dept>>]");
cs=ts;
SchemaObject d_name= cs.getSchemaObject("<<dept,d_name>>");
ts = cs.applyAddTransformation(column,
                          new Object[] {dept,"avgSalary"},
                          "[{'comp',avg[s|{n,s}<-<<comp,salary>>]}," +
                          "{'math',avg[s|{n,s}<-<<math,salary>>]}]");
```

C.1.2 Package uk.ac.bbk.automed.dataWarehousing.util

This package includes the utilities used in the data warehousing toolkit. It has three main classes: QueryDecomposer, IQLEvaluator4DW and Tools4DW.

Class QueryDecomposer

QueryDecomposer class is the implementation of the rules used to decompose a general IQL^c query into a sequence of SIQL queries, as described in Section 5.2.

The public static method queryDecomposer(String `IQLquery`, int `queryNumber`) is used to decompose the string argument `IQLquery` (an IQL^c query represented as a string) and returns an ArrayList object containing the sequence of resulting SIQL queries which are also string objects. The argument `queryNumber` (an integer) is used to generate unique query identifiers when we use this method to decompose successive IQL^c queries. This method creates variables of the form `$Query_queryNumber_i` to express the sub-queries of an IQL^c query.

For example, the list of IQL^c queries:

```
v1 = distinct (D3 -- D4)
v2 = (D1 -- D2) ++ v1
```

is decomposed into following SIQL queries:

```
$Query_1_1 = D3 -- D4
v1         = distinct $Query_1_1
$Query_2_1 = D1 -- D2
v2         = $Query_2_1 ++ v1
```

Class IQLEvaluator4DW

As described in Section 2.2.3, AutoMed's Global Query Processor (GQP) can be used to evaluate an IQL^c query over a global schema in the case of a virtual data integration scenario. The process of evaluating a query over a virtual global schema includes: Query Reformulation, Query Optimisation, Query Annotation and Query Evaluation. There are two limitations of using the AutoMed GQP in our data lineage tracing algorithms:

Firstly, in a data warehouse environment, the global schema will be materialised. The AutoMed GQP is designed for virtual data integration scenarios and does not consider materialised data. Whether the global schema is materialised or not, the AutoMed GQP recomputes the extent of the global schema constructs from the data sources. Using the Query Evaluator directly on materialised data is achieved by the IQLEvaluator4DW class.

The second limitation of the AutoMed GQP is that it can evaluate queries over the constructs of just one schema. For example, the GQP cannot evaluate an IQL^c query ⟨⟨math, name⟩⟩ ++ ⟨⟨comp, name⟩⟩ if the construct ⟨⟨math, name⟩⟩ appears in a source schema and the construct ⟨⟨comp, name⟩⟩ in the global schema. However, in our DLT algorithms, constructs of the source and intermediate schemas frequently appear in the same tracing query. Evaluating IQL^c queries involving constructs from multiple schemas is also achieved by IQLEvaluator4DW class.

The approaches to achieve above two functionalities are as follows:

Firstly, a new Query Reformulation class QueryReformulator4DW inheriting the QueryReformulator class in the AutoMed API has been created. In QueryReformulator4DW, we gather all materialised schema constructs (in the data sources and in the intermediate and global schemas) into a list considered by the reformulation procedure so that it does not replace materialised constructs within the GAV view definitions over the source schema constructs.

Secondly, if there is a virtual construct of an intermediate schema appearing in an IQL^c query, we use the QueryReformulator super class in the AutoMed API to compute its extent by treating the virtual intermediate schema as the global schema.

Class Tools4DW

This class consists of several lower-level methods used by the data warehousing packages. For example, GetIQLSource obtains the names of the schema constructs appearing in an IQL^c query and getQueryType obtains the action type of an IQL^c query.

C.1.3 Package uk.ac.bbk.automed.dataWarehousing.dlt

This package contains the class Lineage, which is the data structure storing lineage data; the class TransfStep, which is the data structure storing transformation steps; the class DataLineageTracing, which is the implementation of the generalised DLT algorithm descried in Chapter 6; and the class DemoDLT, giving an example of using the DLT package.

Class Lineage

The Lineage class has six private attributes which are used to store the information of the lineage data (note that ASG (Abstract Syntax Graph) is the data structure used in the AutoMed GQP for representing IQL queries):

- (ASG)`lineageData`, can be a collection storing materialised lineage data, or, if the lineage data is virtual, it will be `null`;
- (String)`construct`, the name of the schema construct containing the lineage data;
- (boolean)`isVirtualData`, stating if the lineage data is virtual or not;
- (boolean)`isVirtualConstruct`, stating if the construct is virtual or not;
- (String)`eleStruct`, describing the structure of the data in the extent of the schema construct; and
- (String[])`constraint`, expressing the constraints to derive the lineage data from the schema construct if the construct is virtual.

Public non-static methods in this class such as getLineageData(), getConstruct(), isVirtualData(), isVirtualConstruct(), getEleStruct() and getConstraint() are used to obtain the content of the above private attributes.

Class TransfStep

The TransfStep class contains six private attributes storing the information of the transformation steps:

- (String)`action`, which may be ″`add`″, ″`del`″, ″`rename`″, ″`extend`″ and ″`contract`″;

- (String)`query`, the query used in the transformation step;
- (String)`result`, the name of the schema construct created or deleted by the transformation step;
- (boolean)`vResult`, showing if the result construct is virtual or not;
- (ArrayList)`sources`, containing all schema construct names appearing in the query; and
- (boolean[])`vSources`, showing which source constructs in the `sources` collection are virtual.

Public non-static methods such as getAction(), getQuery(), getResult(), isVResult(), getSources() and getVSources() are used to obtain the content of the above private attributes.

In addition, there are two static methods available in this class which can be used to obtain the transfStep objects between a given source and global schema. In particular, the method ArrayList getTransfSteps(String `sName`, String `gName`) results in an ArrayList collection containing transfStep objects expressing the general transformation pathway (may contain general IQL^c queries) between the two schemas, `sName` and `gName`. The method ArrayList getSimpleTransfSteps(String `sName`, String `gName`) results in an ArrayList collection containing transfStep objects expressing the decomposed transformation pathway (all general IQL^c queries in the general transformation pathway have been decomposed into SIQL queries) between the schema `sName` and `gName`.

Class DataLineageTracing

In the DataLineageTracing class, the method DLT4AStep(Lineage `tt`, TransfStep `ts`) is used to obtain the lineage of a single tracing tuple `tt` along a single transformation step `ts`, while the methods oneDLT4APath(Lineage `tt`, ArrayList `tp`) and listDLT4APath(ArrayList `tts`, ArrayList `tp`) are respectively used to obtain the lineage of a single tracing tuple `tt` or a bag of tracing tuples `tts` along the transformation pathway `tp`.

The constructor of this class is DataLineageTracing(Schema `sSchema`,Schema `tSchema`), in which `sSchema` and `tSchema` are two Schema objects denoting the source and target schemas. Once a DataLineageTracing object, `dlt`, is created, the simple transformation steps between the source and target schemas are also generated and stored. The public non-static method `dlt`.getTransformationSteps() is then used to obtain the generated simple transformation steps between the given source and target schemas, and the public non-static methods `dlt`.getDataLineageOf(Lineage `lp`) and `dlt`.getDataLineageOf(ArrayList `lpList`) are used to obtain the lineage of the tracing data.

Class DemoDLT

The DemoDLT class gives an example of using the DLT toolkit for tracing data lineage along an AutoMed transformation pathway. In particular, after creating the AutoMed metadata, the DLT process is accomplished by the following three steps:

1. Getting the source and global schemas by using the Schema.getSchema(String `schemaName`) method provided by the AutoMed API. For example:

   ```
   Schema s_sou = Schema.getSchema("rel_source");
   Schema s_tar = Schema.getSchema("rel_global");
   ```

2. Creating a DataLineageTracing object, dlt:

   ```
   DataLineageTracing dlt = new DataLineageTracing(s_sou,s_tar);
   ```

3. Giving the tracing tuple and tracing its data lineage. For example, for tracing tuple `{'M01',1000}` in the construct $\langle\langle \mathsf{person}, \mathsf{salary} \rangle\rangle$ of the target schema `"rel_global"`, the necessary code is :

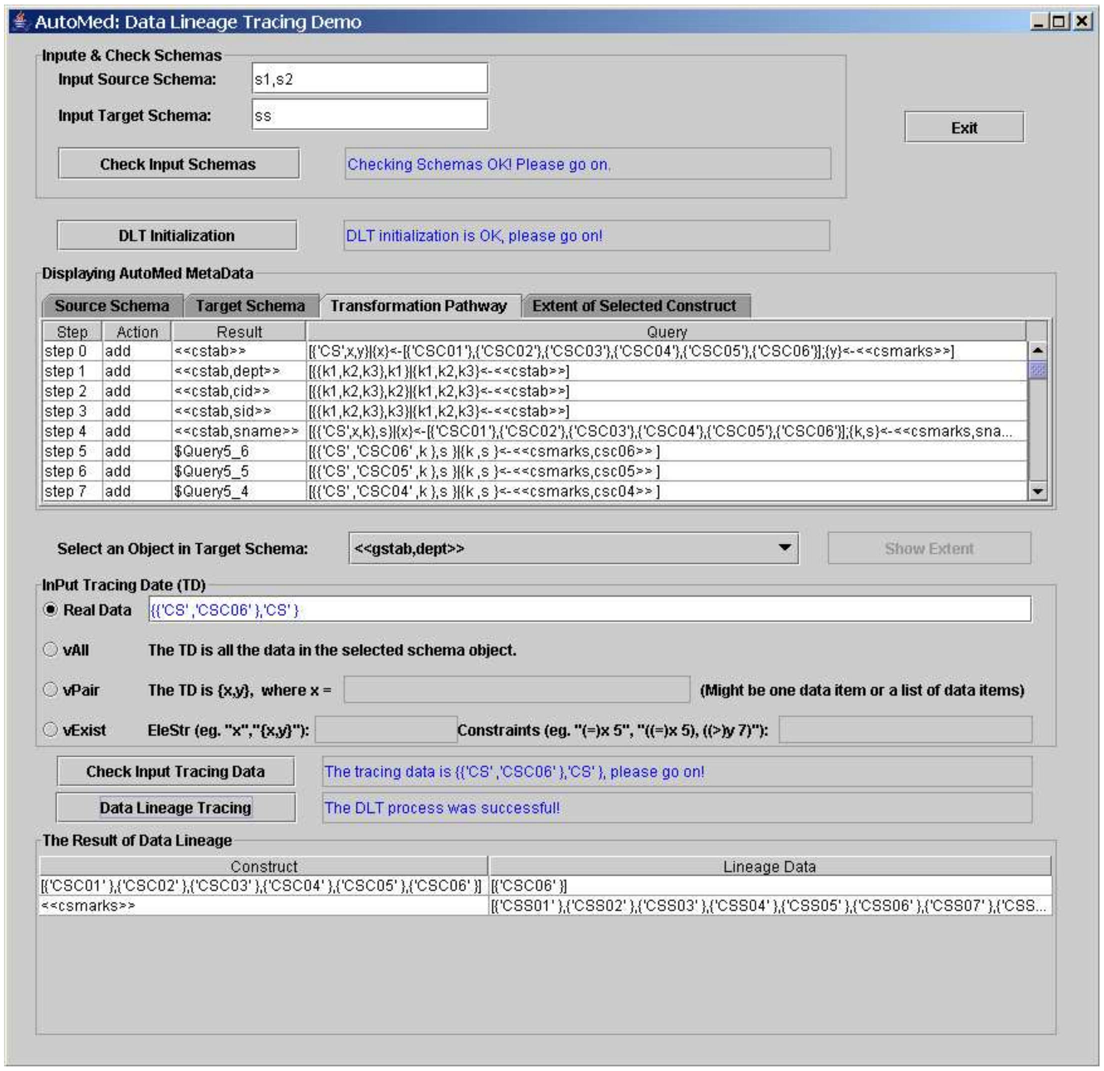
   ```
   Lineage tt = new Lineage(
                  new ASG("{'M01',1000}"),"<<person,salary>>");
   ArrayList lineageData = new ArrayList();
   lineageData = dlt.getDataLineageOf(tt);
   Lineage.printLineageList(lineageData);
   ```

C.2 Data Lineage Tracing GUI

In this section, we describe a GUI supporting our data lineage tracing process, and show how our DLT process can be applied in both materialised and virtual data integration scenarios. We also show how the DLT GUI can be used as a tool for browsing schemas, data and lineage information.

C.2.1 The DLT GUI

Figure C.1 illustrates the DLT GUI. Given the names of the source schemas, *e.g.* `s1` and `s2`, and target schema, *e.g.* `ss`, the "Check Input Schema" button is used to check whether the input schema names are defined in the AutoMed Schemas and Transformations Repository (STR). Then the "DLT Initialization" button is used to initialise the DLT process, which consists of three main steps: obtaining the source and target schemas from the AutoMed STR and listing their constructs; obtaining the transformation pathway between the source and target schemas,



AutoMed: Data Lineage Tracing Demo

Inpute & Check Schemas

Input Source Schema: s1,s2

Input Target Schema: ss

Check Input Schemas — Checking Schemas OK! Please go on.

Exit

DLT Initialization — DLT initialization is OK, please go on!

Displaying AutoMed MetaData

Source Schema | Target Schema | Transformation Pathway | Extent of Selected Construct

Step	Action	Result	Query
step 0	add	<<cstab>>	[{'CS',x,y}|{x}<-[{'CSC01'},{'CSC02'},{'CSC03'},{'CSC04'},{'CSC05'},{'CSC06'}];{y}<-<<csmarks>>]
step 1	add	<<cstab,dept>>	[{{k1,k2,k3},k1}|{k1,k2,k3}<-<<cstab>>]
step 2	add	<<cstab,cid>>	[{{k1,k2,k3},k2}|{k1,k2,k3}<-<<cstab>>]
step 3	add	<<cstab,sid>>	[{{k1,k2,k3},k3}|{k1,k2,k3}<-<<cstab>>]
step 4	add	<<cstab,sname>>	[{{'CS',x,k},s}|{x}<-[{'CSC01'},{'CSC02'},{'CSC03'},{'CSC04'},{'CSC05'},{'CSC06'}];{k,s}<-<<csmarks,sna...
step 5	add	$Query5_6	[{{'CS' ,'CSC06' ,k },s }|{k ,s }<-<<csmarks,csc06>>]
step 6	add	$Query5_5	[{{'CS' ,'CSC05' ,k },s }|{k ,s }<-<<csmarks,csc05>>]
step 7	add	$Query5_4	[{{'CS' ,'CSC04' ,k },s }|{k ,s }<-<<csmarks,csc04>>]

Select an Object in Target Schema: <<gstab,dept>> — Show Extent

InPut Tracing Date (TD)

● Real Data {{'CS' ,'CSC06' },'CS' }

○ vAll The TD is all the data in the selected schema object.

○ vPair The TD is {x,y}, where x = (Might be one data item or a list of data items)

○ vExist EleStr (eg. "x","{x,y}"): Constraints (eg. "(=)x 5", "((=)x 5), ((>)y 7)"):

Check Input Tracing Data — The tracing data is {{'CS' ,'CSC06' },'CS' }, please go on!

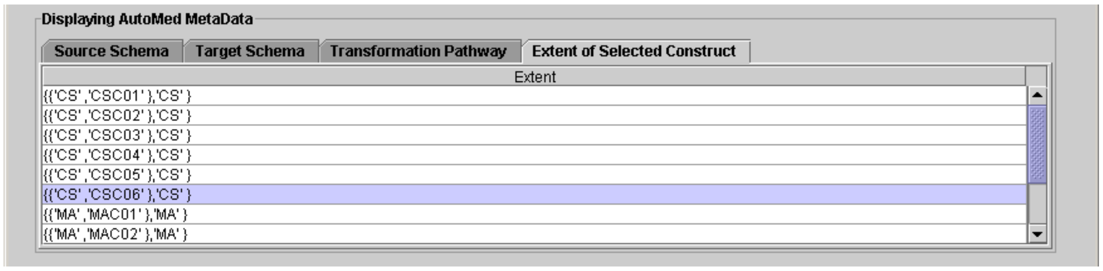
Data Lineage Tracing — The DLT process was successful!

The Result of Data Lineage

Construct	Lineage Data
[{'CSC01' },{'CSC02' },{'CSC03' },{'CSC04' },{'CSC05' },{'CSC06' }]	[{'CSC06' }]
<<csmarks>>	[{'CSS01' },{'CSS02' },{'CSS03' },{'CSS04' },{'CSS05' },{'CSS06' },{'CSS07' },{'CSS...

Figure C.1: The Data Lineage Tracing GUI

decomposing it into a simple transformation pathway and listing the pathway (illustrated in Figure C.1); and initialising a `DataLineageTracing` object.



Extent
{{'CS','CSC01'},'CS'}
{{'CS','CSC02'},'CS'}
{{'CS','CSC03'},'CS'}
{{'CS','CSC04'},'CS'}
{{'CS','CSC05'},'CS'}
{{'CS','CSC06'},'CS'}
{{'MA','MAC01'},'MA'}
{{'MA','MAC02'},'MA'}

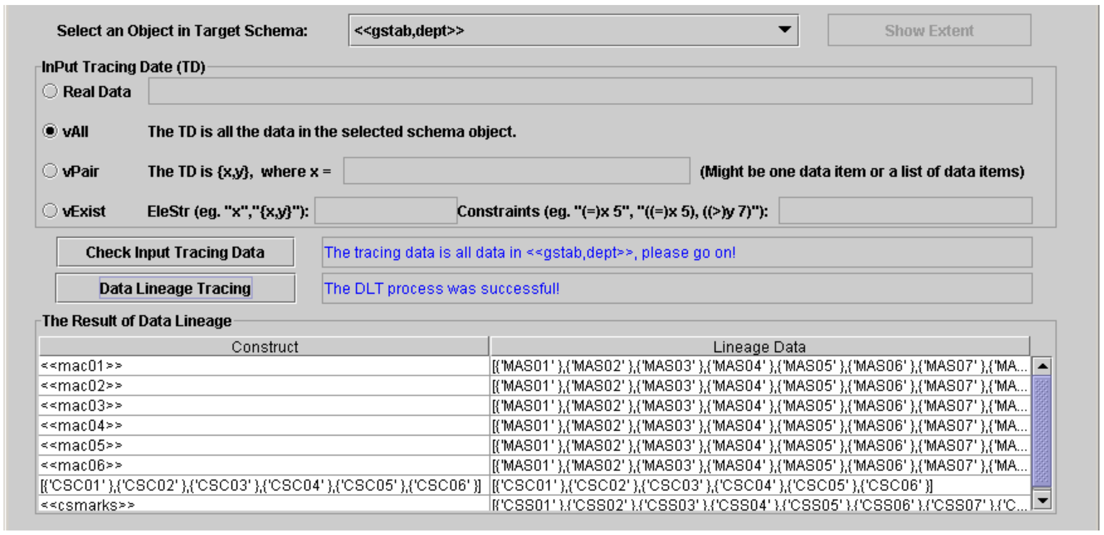
Figure C.2: The Extent of Selected Construct

After DLT initialisation, the "Show Extent" button can be used to extract the extent of the selected construct in the target schema and show it in the "Extent of Selected Construct" field (as in Figure C.2). The displayed data items can then be selected as the tracing tuples of the DLT process.

More generally, four kinds of tracing tuples that may be input[1]: RealData, which is one or more data items selected from the extent of the target schema construct (as in Figure C.1); vAll, where the tracing data is all data in the selected target construct (as in Figure C.3); vPair, where the tracing data is a pair such as `{x,y}` where the extent of `x` is indicated (as in Figure C.4); and vExist, where the tracing data is an arbitrary pattern, such as `{{d,c},x}`, and constraints over its variables can also be specified, such as "`(>=) x 67`" (as in Figure C.5).

Once a tracing tuple is selected, the "Check Input Tracing Data" button semantically checks the input tracing tuple, and the "Data Lineage Tracing" button finally computes the lineage of the tracing tuple.

[1]These correspond to real lineage data and the three kinds of virtual lineage data, $\{\texttt{any}, \texttt{true}\}$, $(\{\texttt{x}, \texttt{y}\}, \texttt{x} = \texttt{a})$ and $(\texttt{p1}, \texttt{p2} = \texttt{t})$, described in Chapter 6.


Select an Object in Target Schema: <<gstab,dept>>

InPut Tracing Date (TD)

Real Data [✓ vAll] vPair vExist

vAll: The TD is all the data in the selected schema object.

vPair: The TD is {x,y}, where x = (Might be one data item or a list of data items)

vExist: EleStr (eg. "x","{x,y}"): Constraints (eg. "(=)x 5", "((=)x 5), ((>)y 7)"):

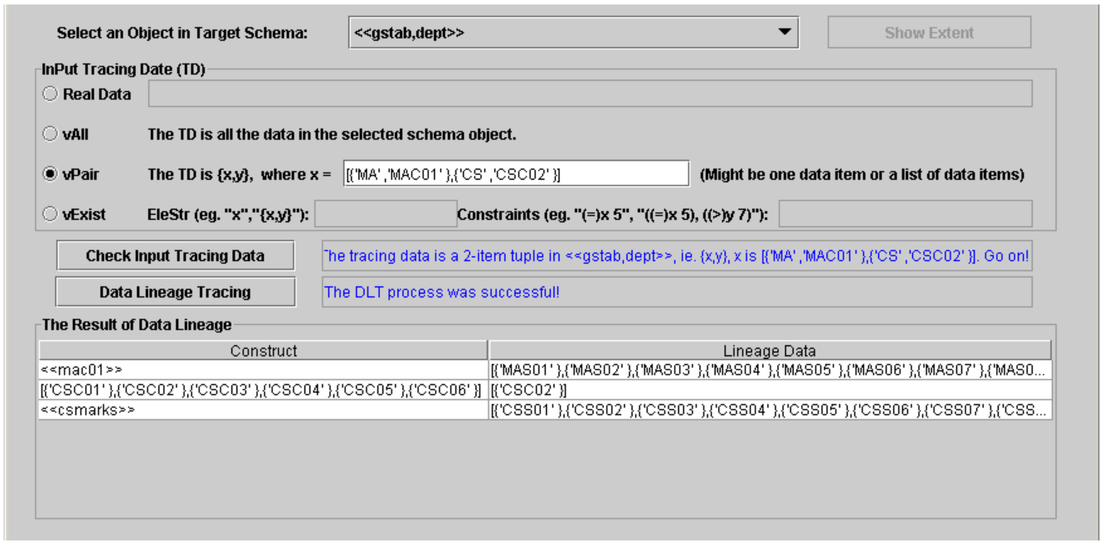
Check Input Tracing Data: The tracing data is all data in <<gstab,dept>>, please go on!

Data Lineage Tracing: The DLT process was successful!

The Result of Data Lineage

Construct	Lineage Data
<<mac01>>	[{'MAS01' },{'MAS02' },{'MAS03' },{'MAS04' },{'MAS05' },{'MAS06' },{'MAS07' },{'MA...
<<mac02>>	[{'MAS01' },{'MAS02' },{'MAS03' },{'MAS04' },{'MAS05' },{'MAS06' },{'MAS07' },{'MA...
<<mac03>>	[{'MAS01' },{'MAS02' },{'MAS03' },{'MAS04' },{'MAS05' },{'MAS06' },{'MAS07' },{'MA...
<<mac04>>	[{'MAS01' },{'MAS02' },{'MAS03' },{'MAS04' },{'MAS05' },{'MAS06' },{'MAS07' },{'MA...
<<mac05>>	[{'MAS01' },{'MAS02' },{'MAS03' },{'MAS04' },{'MAS05' },{'MAS06' },{'MAS07' },{'MA...
<<mac06>>	[{'MAS01' },{'MAS02' },{'MAS03' },{'MAS04' },{'MAS05' },{'MAS06' },{'MAS07' },{'MA...
[{'CSC01' },{'CSC02' },{'CSC03' },{'CSC04' },{'CSC05' },{'CSC06' }]	[{'CSC01' },{'CSC02' },{'CSC03' },{'CSC04' },{'CSC05' },{'CSC06' }]
<<csmarks>>	[{'CSS01' },{'CSS02' },{'CSS03' },{'CSS04' },{'CSS05' },{'CSS06' },{'CSS07' },{'C...

Figure C.3: Tracing Data Lineage of vAll


Select an Object in Target Schema: <<gstab,dept>>

InPut Tracing Date (TD)

Real Data vAll [✓ vPair] vExist

vAll: The TD is all the data in the selected schema object.

vPair: The TD is {x,y}, where x = [{'MA' ,'MAC01' },{'CS' ,'CSC02' }] (Might be one data item or a list of data items)

vExist: EleStr (eg. "x","{x,y}"): Constraints (eg. "(=)x 5", "((=)x 5), ((>)y 7)"):

Check Input Tracing Data: 'he tracing data is a 2-item tuple in <<gstab,dept>>, ie. {x,y}, x is [{'MA' ,'MAC01' },{'CS' ,'CSC02' }]. Go on!

Data Lineage Tracing: The DLT process was successful!

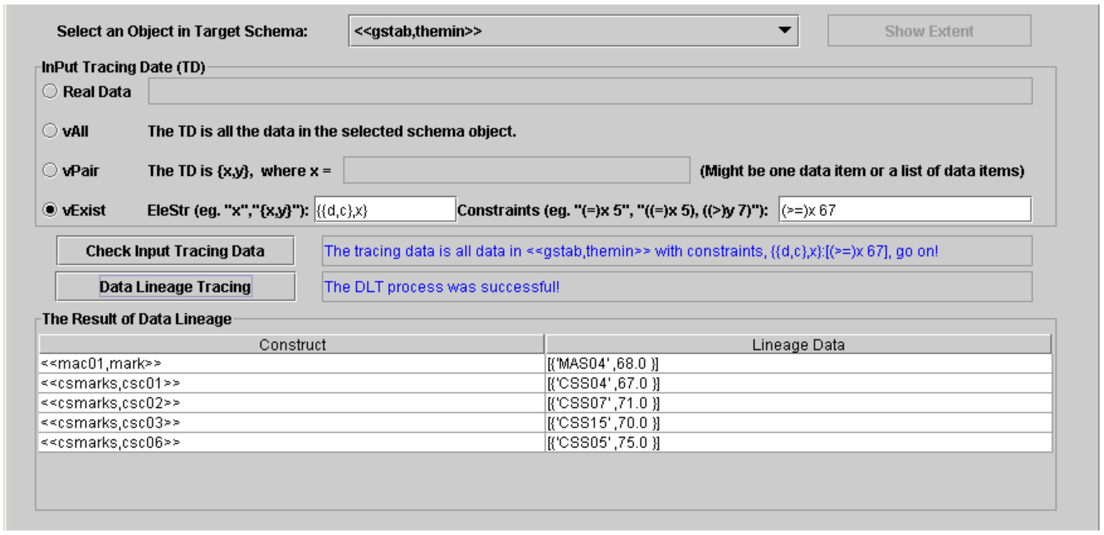
The Result of Data Lineage

Construct	Lineage Data
<<mac01>>	[{'MAS01' },{'MAS02' },{'MAS03' },{'MAS04' },{'MAS05' },{'MAS06' },{'MAS07' },{'MAS0...
[{'CSC01' },{'CSC02' },{'CSC03' },{'CSC04' },{'CSC05' },{'CSC06' }]	[{'CSC02' }]
<<csmarks>>	[{'CSS01' },{'CSS02' },{'CSS03' },{'CSS04' },{'CSS05' },{'CSS06' },{'CSS07' },{'CSS...

Figure C.4: Tracing Data Lineage of vPair

C.2.2 DLT in Materialised Data Integration

In materialised data integration scenarios, both the source and target schemas are materialised *e.g.* in the example of Section 4.2 the data source schemas `s1,s2` and the global schema `ss` are all materialised. The figures of Section C.2.1 illustrated


Select an Object in Target Schema: <<gstab,themin>>

InPut Tracing Date (TD)

○ Real Data

○ vAll The TD is all the data in the selected schema object.

○ vPair The TD is {x,y}, where x = (Might be one data item or a list of data items)

● vExist EleStr (eg. "x","{x,y}"): {{d,c},x} Constraints (eg. "(=)x 5", "((=)x 5), ((>)y 7)"): (>=)x 67

The tracing data is all data in <<gstab,themin>> with constraints, {{d,c},x}:[(>=)x 67], go on!

The DLT process was successful!

The Result of Data Lineage

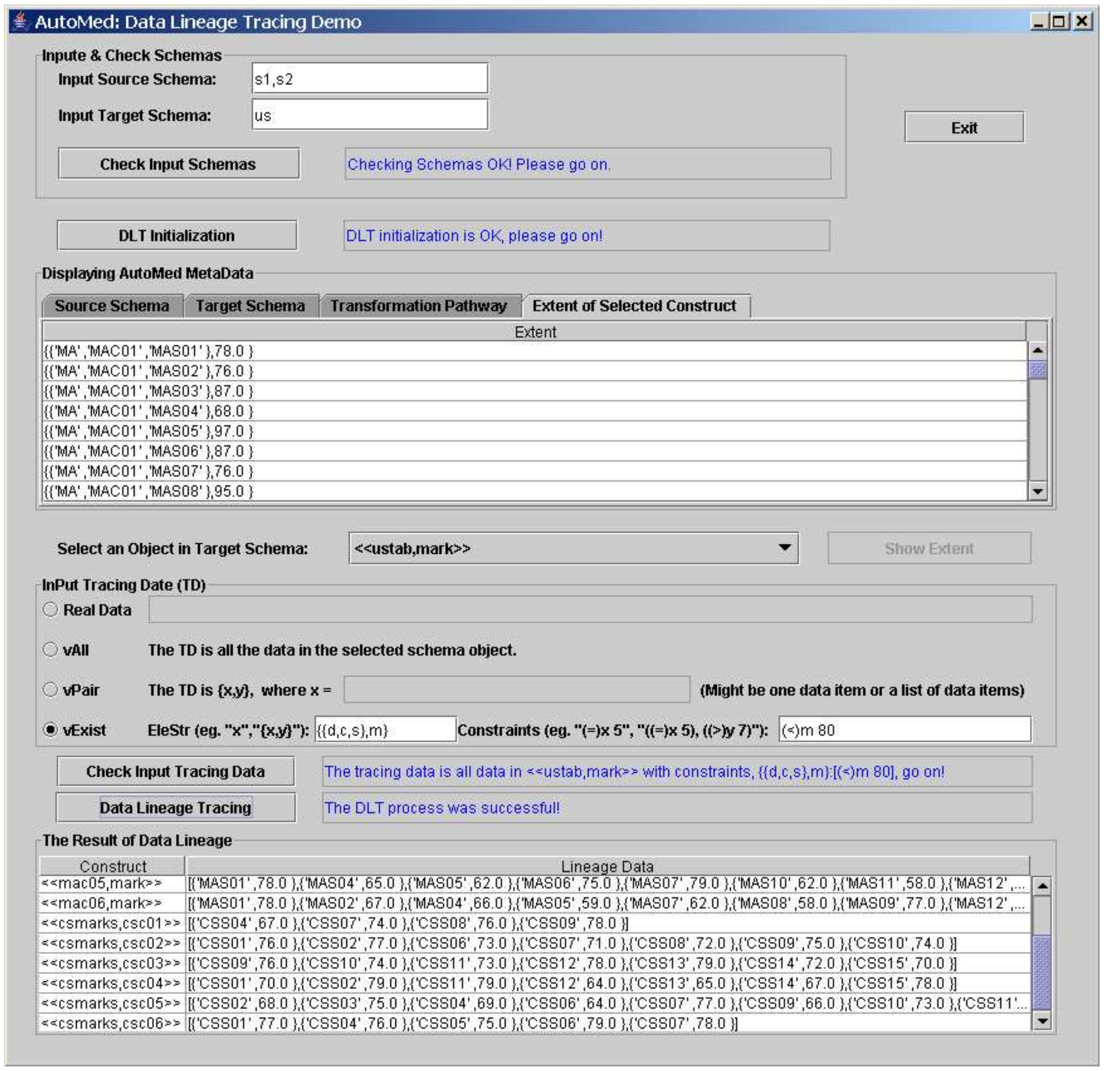
Construct	Lineage Data
<<mac01,mark>>	[{'MAS04' ,68.0 }]
<<csmarks,csc01>>	[{'CSS04' ,67.0 }]
<<csmarks,csc02>>	[{'CSS07' ,71.0 }]
<<csmarks,csc03>>	[{'CSS15' ,70.0 }]
<<csmarks,csc06>>	[{'CSS05' ,75.0 }]

Figure C.5: Tracing Data Lineage of vExist

how the DLT GUI can be used in a materialised data integration scenario.

C.2.3 DLT in Virtual Data Integration

In virtual data integration scenarios, the target and all intermediate schemas are virtual. Figure C.6 illustrates how the DLT GUI can be used in a virtual data integration scenario, in which the input target schema `us` is a virtual one. We assume the same framework described as in the example of Section 4.2 and use the virtual schema `US` as the target schema. In Figure C.6, the lineage of the vExist tracing data, $\langle\langle\mathsf{ustab}, \mathsf{mark}\rangle\rangle|$`({{d,c,s},m}, (=) m 80)`, is computed. The lineage of other kinds of tracing data such as RealData, vAll and vPair are also traceable in this virtual data integration scenario.



AutoMed: Data Lineage Tracing Demo

Inpute & Check Schemas

Input Source Schema: s1,s2

Input Target Schema: us

Check Input Schemas — Checking Schemas OK! Please go on.

Exit

DLT Initialization — DLT initialization is OK, please go on!

Displaying AutoMed MetaData

Source Schema | Target Schema | Transformation Pathway | Extent of Selected Construct

Extent
{{'MA' ,'MAC01' ,'MAS01' },78.0 }
{{'MA' ,'MAC01' ,'MAS02' },76.0 }
{{'MA' ,'MAC01' ,'MAS03' },87.0 }
{{'MA' ,'MAC01' ,'MAS04' },68.0 }
{{'MA' ,'MAC01' ,'MAS05' },97.0 }
{{'MA' ,'MAC01' ,'MAS06' },87.0 }
{{'MA' ,'MAC01' ,'MAS07' },76.0 }
{{'MA' ,'MAC01' ,'MAS08' },95.0 }

Select an Object in Target Schema: <<ustab,mark>> Show Extent

InPut Tracing Date (TD)

○ Real Data

○ vAll — The TD is all the data in the selected schema object.

○ vPair — The TD is {x,y}, where x = (Might be one data item or a list of data items)

● vExist — EleStr (eg. "x","{x,y}"): {{d,c,s},m} Constraints (eg. "(=)x 5", "((=)x 5), ((>)y 7)"): (<)m 80

Check Input Tracing Data — The tracing data is all data in <<ustab,mark>> with constraints, {{d,c,s},m}:[(<)m 80], go on!

Data Lineage Tracing — The DLT process was successful!

The Result of Data Lineage

Construct	Lineage Data
<<mac05,mark>>	[{'MAS01' ,78.0 },{'MAS04' ,65.0 },{'MAS05' ,62.0 },{'MAS06' ,75.0 },{'MAS07' ,79.0 },{'MAS10' ,62.0 },{'MAS11' ,58.0 },{'MAS12' ,...
<<mac06,mark>>	[{'MAS01' ,78.0 },{'MAS02' ,67.0 },{'MAS04' ,66.0 },{'MAS05' ,59.0 },{'MAS07' ,62.0 },{'MAS08' ,58.0 },{'MAS09' ,77.0 },{'MAS12' ,...
<<csmarks,csc01>>	[{'CSS04' ,67.0 },{'CSS07' ,74.0 },{'CSS08' ,76.0 },{'CSS09' ,78.0 }]
<<csmarks,csc02>>	[{'CSS01' ,76.0 },{'CSS02' ,77.0 },{'CSS06' ,73.0 },{'CSS07' ,71.0 },{'CSS08' ,72.0 },{'CSS09' ,75.0 },{'CSS10' ,74.0 }]
<<csmarks,csc03>>	[{'CSS09' ,76.0 },{'CSS10' ,74.0 },{'CSS11' ,73.0 },{'CSS12' ,78.0 },{'CSS13' ,79.0 },{'CSS14' ,72.0 },{'CSS15' ,70.0 }]
<<csmarks,csc04>>	[{'CSS01' ,70.0 },{'CSS02' ,79.0 },{'CSS11' ,79.0 },{'CSS12' ,64.0 },{'CSS13' ,65.0 },{'CSS14' ,67.0 },{'CSS15' ,78.0 }]
<<csmarks,csc05>>	[{'CSS02' ,68.0 },{'CSS03' ,75.0 },{'CSS04' ,69.0 },{'CSS06' ,64.0 },{'CSS07' ,77.0 },{'CSS09' ,66.0 },{'CSS10' ,73.0 },{'CSS11'...
<<csmarks,csc06>>	[{'CSS01' ,77.0 },{'CSS04' ,76.0 },{'CSS05' ,75.0 },{'CSS06' ,79.0 },{'CSS07' ,78.0 }]

Figure C.6: Tracing Data Lineage with a Virtual Schema

C.2.4 A Tool for Browsing Schemas, Data and Lineage Information

The DLT GUI can be used to browse the extent of both materialised and virtual target schemas, as well as the constructs of these schemas and the lineage of their data.

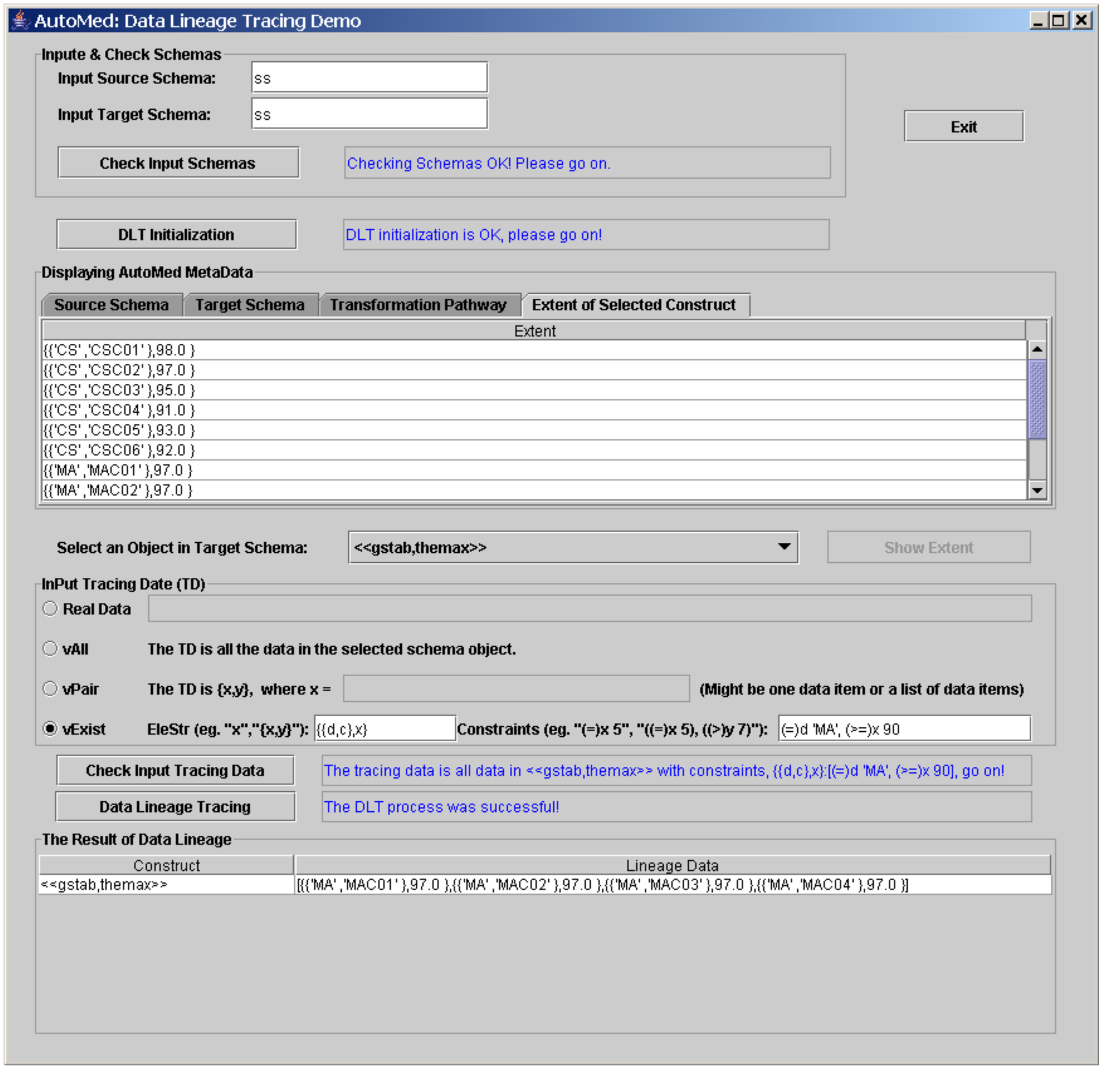
If we define the input source and target schemas as being the same schema, the DLT GUI can be used as a simple query engine over this schema. For example, in Figure C.7, both the input source and target schemas are `ss`. If the tracing data is vExist data, $\langle\langle \mathsf{gstab}, \mathsf{themax} \rangle\rangle$`|({{d,c},x},[(=) d 'MA',(>=) x 80])`, the computed lineage data is actually equivalent to applying the IQL^c query [`{{d,c},x}`| `{{d,c},x}` $\leftarrow$ $\langle\langle \mathsf{gstab}, \mathsf{themax} \rangle\rangle$; `(=) d 'MA';(>=) x 80`] to the schema `ss`.

C.3 Discussion

In this appendix, we have discussed a set of data warehousing packages and API for the AutoMed toolkit, which implement the generalised DLT algorithm described in Chapter 6. Currently, the data warehousing toolkit consists of three packages: dataWarehousing.dlt, dataWarehousing.util and dataWarehousing.DW-Example.

We have given a data integration scenario and example to illustrate how our DLT process and GUI can be applied, both in materialised and virtual data integration settings. We have also discussed how the DLT GUI can be used as a tool for browsing schemas, data and lineage information.

In Section 6.6.1 of Chapter 6 and Section 7.4.1 of Chapter 7, we discussed how to extend our DLT and IVM algorithms to handle queries beyond IQL^c. This would allow our DLT process to go back all the way to the data source



Input Source Schema: ss

Input Target Schema: ss

Checking Schemas OK! Please go on.

DLT initialization is OK, please go on!

Extent
{{'CS' ,'CSC01' },98.0 }
{{'CS' ,'CSC02' },97.0 }
{{'CS' ,'CSC03' },95.0 }
{{'CS' ,'CSC04' },91.0 }
{{'CS' ,'CSC05' },93.0 }
{{'CS' ,'CSC06' },92.0 }
{{'MA' ,'MAC01' },97.0 }
{{'MA' ,'MAC02' },97.0 }

Select an Object in Target Schema: <<gstab,themax>>

○ Real Data

○ vAll The TD is all the data in the selected schema object.

○ vPair The TD is {x,y}, where x = (Might be one data item or a list of data items)

● vExist EleStr (eg. "x","{x,y}"): {{d,c},x} Constraints (eg. "(=)x 5", "((=)x 5), ((>)y 7)"): (=)d 'MA', (>=)x 90

The tracing data is all data in <<gstab,themax>> with constraints, {{d,c},x}:[(=)d 'MA', (>=)x 90], go on!

The DLT process was successful!

Construct	Lineage Data
<<gstab,themax>>	[{{'MA' ,'MAC01' },97.0 },{{'MA' ,'MAC02' },97.0 },{{'MA' ,'MAC03' },97.0 },{{'MA' ,'MAC04' },97.0 }]

Figure C.7: Browsing Schemas and Data Information

schemas before single-source cleansing, and would similarly allow our IVM process to maintain materialised warehouse data according to updates to the data source schemas. The implementation of these extensions is an area of future work.

Glossary

BAV Both-as-view data integration approach, 17

CDM Conceptual data model, 44

DLT Data lineage tracing, 100

GAV Global-as-view data integration approach, 16

GQP Global Query Processor, 59

HDM Hypergraph-based data model, 45

IQL Intermediate query language, 54

IQL^c A subset of IQL, 100

IVM Incremental view maintenance, 171

LAV Local-as-view data integration approach, 16

MDR The AutoMed Model Definitions Repository, 61

SIQL Simple intermediate query language, 105

STR The AutoMed Schemas and Transformations Repository, 61