Bridging Decision Applications and Multidimensional Databases

by

Fatemeh Nargesian

Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements

For the M.Sc. degree in

Computer Science

School of Information Technology and Engineering

University of Ottawa

Abstract

Data warehouses were envisioned to facilitate analytical reporting and data visualization by providing a model for the flow of data from operational databases to decision support environments. Decision support environments provide a multidimensional conceptual view of the underlying data warehouse, which is usually stored in relational DBMSs. Typically, there is an impedance mismatch between this conceptual view — shared also by all decision support applications accessing the data warehouse — and the physical model of the data stored in relational DBMSs. This thesis presents a mapping compilation algorithm in the context of the *Conceptual Integration Model (CIM)* [67] framework. In the CIM framework, the relationships between the conceptual model and the physical model are specified by a set of attribute-to-attribute correspondences. The algorithm compiles these correspondences into a set of mappings that associate each construct in the conceptual model with a query on the physical model. Moreover, the homogeneity and summarizability of data in conceptual models is the key to accurate query answering, a necessity in decision making environments. A data-driven approach to refactor relational models into summarizable schemas and instances is proposed as the solution of this issue. We outline the algorithms and challenges in bridging multidimensional conceptual models and the physical model of data warehouses and discuss experimental results.

Acknowledgements

First and foremost I would like to thank my supervisors, Iluju Kiringa and Liam Peyton. I am grateful to my supervisor Dr. Iluju Kiringa, for providing me with opportunity, ethics, and guidance in the way of innovation and novelty. I am grateful to my co-supervisor, Dr. Liam Peyton, for having provided me with opportunity, ethics, and a sound research methodology. Without their help and support this work would not have been possible. Their guidance and encouragement throughout my research was invaluable. I would also like to thank Dr. Flavio Rizzolo. He collaborated with me on some work and his help and support greatly improved the quality of my work. I also thank the Data Integration research group in NSERC Business Intelligence Network for their helpful comments and contributions. This work was supported by a research grant from the NSERC Business Intelligence Network and by the University of Ottawa through the International Francophone Scholarship. Finally, I would like to thank my parents for their support and encouragement throughout the writing of this thesis.

Contents

List of Tables

List of Figures

[1]http://datawarehouse4u.info/
[2]http://datawarehouse4u.info/
[3]http://datawarehouse4u.info/
[4]http://danlinstedt.com/about/data-vault-basics/

Chapter 1

Introduction

Modern data centric applications manipulate data represented in multiple formats, such as objects in a programming language, constructs in conceptual models, rows in relational databases, or XML structures. Specifically, in Business Intelligence tools, the analytical data in data warehouse is conceptualized as a semantic layer with which users can interact easily. As a result, application developers face a constant challenge of translating data and data access operations across different data representations. A common way of shielding the semantics of the conceptual layer from the intricacies of data manipulation is by using a data access layer that encapsulates the necessary transformations [58]. Typically, a data access layer provides an updatable client-side view that exposes persistent data as constructs in a conceptual model. In the case of multidimensional conceptual models in business intelligence tools, the views do not need to be updatable since the objective is mainly to perform read-only analytical reporting against the underlying physical data warehouse. It is straight forward to hardwire the data access code for a specific application. However, in order to have a general-purpose data access layer, that can be maintained as both the conceptual models and underlying physical data warehouse evolve, a mapping can be used to establish a relationship between the conceptual models

and the physical model of the data warehouse. In a business intelligence application, a mapping can parameterize or automatically generate the logic to transforming data from the physical model to conceptual model (or to generate the queries at the conceptual level into queries against the physical model?). Therefore, mapping-driven data access is of critical importance for query answering through conceptual models.

1.1 Motivation

A data warehouse is a largely static repository which gathers data for analysis. Data warehouses typically contain fact and dimension tables that provide data about different parts of an organization. Business analysts require a clear abstract view of a subset of the data warehouse in order to ask for reports about a specific topic and make strategic decisions. Typically, a multidimensional view of data is designed as an extra layer on top of, or in front of, date warehouses which allows the basic data (stored in fact tables) to be analyzed according to various hierarchies which is appropriate for analysts and decision makers. There is usually a complex semantic impedance mismatch between the physical model of a data warehouse and multidimensional conceptual models. It is not reasonable to expect business analysts to understand the mapping between the two types of models. However, an ad hoc mapping mechanism facilitates reorganizing the conceptual model according to users' needs. In order to have a data access layer that does not require analysts to have complete knowledge about storage details, yet allow them to redesign the conceptual layouts, we can establish a framework with automatic compilation of simple mappings. In such a framework, users specify simple mappings drawn from attributes of a conceptual model to columns of tables. Then at compilation time, these mappings are processed and a new set of views are generated over the data warehouse. These views facilitate the transformation of data between two models.

1.2 The Problem

It is relatively straightforward to manually create a mapping between two schemas that are very smilar, *i.e.*, exposing one table as an object. However, as more semantically distinct structures are defined for store and conceptual models, generating mappings becomes more complex. This is justified by the fact that mapping-driven data access lies at the intersection of two data management problems: (1) Impedance mismatch, which is the problem of accessing persistent storage data through constructs of an abstract model. Its focus is on bridging the gap between two distinct data models, usually by defining a set of mappings. (2) Data integration, which is the problem of providing unified access to heterogeneous data [58]. In order to build a backbone connecting a multidimensional conceptual model to a store model, the impendence mismatch problem can be addressed by creating a set of views which maps conceptual model constructs to database elements. The reality of making this approach is that the soundness and exactness of the data returned by these views are based on the characteristics of the underlying data, such as summarizability [32]. Summarizability guarantees the correctness of aggregated values in the data warehouse resulted by performing simple roll-up operations on the data existing in dimensions. Satisfying summarizability implies doing data reshaping. Therefore, in a multidimensional conceptual integration framework impedance mismatch and data integration problems go hand in hand.

1.3 Our Approach

In this thesis, we address the problem of impedance mismatch between multidimensional databases and conceptual models as well as reshaping the data. We make the following contributions.

First, we present an algorithm for automatically creating mappings between conceptual and logical multidimensional models. The algorithm compiles simple user-provided attribute-to-attribute mappings into complex views over the multidimensional store model. Our method represents the store schema and its constraints as a graph and performs graph summarization and traversal algorithms in order to find views with minimum necessary join operations over store schema. In addition, evaluation results show the viability of the compilation algorithm with syntectic and real-world data.

Second, we propose a data-based algorithm that deals with the heterogeneity and summarizability issues in the multidimensional model. The condition of creating exact views by mapping compilation algorithm is to have summarizable stored data. We generate a summarizable schema and instance of data warehouse by applying the bisimulation algorithm [38] on data stored in dimensions. This dimension refactoring is considered as a preprocessing phase for mapping compilation algorithm. We also theoretically study the soundness and complexity of this algorithm and point out its feasibility according to the size of real data warehouses.

1.4 The Methodology and Outline

The methodology we employed during our work was problem formulation, resolution criteria specification, then development and evaluation of our proposed solution to the problem by running experiments on benchmarks. In particular, we took the following steps:

1. Identify and analyze problem (bridging the gap between decision applications and multidimensional databases)

2. Review literature and finding the most appropriate framework (Conceptual Inte-

gration Model (CIM)) in which the problem can be defined

3. Customizing the existing framework (CIM) for our problem
4. Review literature and industry approaches to similar problems (mapping objects in applications to relational databases)
5. Identify the format of the output (views over data warehouse) and its characteristics (view with minimum number of joins; soundness and exactness of views)
6. Develop an algorithm which generates views with minimum number of joins and in polynomial time
7. Perform case study (DreamHome, Infection Control and TPC-H) to evaluate the proposed algorithm
8. Study the soundness and exactness of the results
9. Identify the conditions of exact solutions based on the characteristics of inputs (well-formed data warehouses and summarizable dimension models)
10. Propose a general algorithm for refactoring dimension models into summarizable models (thus satisfying the condition for having exact views)
11. Draw conclusions and identify future work

The thesis is organized as follows: Chapter 2 provides background on existing conceptual multidimensional models and physical models of data warehouses. Since the objective of this thesis is to solve the problem of impedance mismatch between multidimensional conceptual models and data warehouse physical models, related approaches proposed for similar problems in the literature are discussed. General background on business intelligence applications are also covered in this chapter. In Chapter 3, we

present the framework in which our approach for impedance mismatch is used. We also describe the preprocessing phase that is essential in order to have desirable mappings between conceptual and store models. In Chapter 4, we describe our approach to mapping compilation and view generation. Then, Chapter 5 presents the results of experiments we have performed based on our proposed approach including the creation of benchmarks used in the evaluation. Finally, in Chapter 6 we present our conclusions with a brief discussion of possibilities for future work.

Chapter 2

Related Work

2.1 Multidimensional Conceptual Models

Conceptual modeling is a technique of data modeling, which is independent of implementation details, such as data storage [40, 37]. A conceptual model represents the meaning of concepts in a domain and the relationships between them in order to facilitate discussing the problem. It clarifies the meaning of ambiguous terms, and ensures that problems with different interpretations of the terms and concepts do not occur. After finalizing the conceptual model, it becomes a basis for subsequent development of applications. In database modeling, a conceptual model can be used as basis of designing the underlying database in a top-down manner. However, in a bottom-up modeling, a conceptual model can be created for an existing database in order to provide users with a comprehensive view of database. This facilitates modeling the reports users may request the querying application.

Data warehouses meet an organization's need for reliable, consolidated, unique and integrated analysis and reporting, through providing data at different levels of aggregation [40]. The conceptual model of a data warehouse can assist users in understanding

how the data is organized as a multidimensional database, and what reports they can request. Consequently, a conceptual model should cover all the concepts in a data warehouse, such as facts, measures, levels, hierarchies, and dimensions. In this section, we go over existing conceptual models for representing multidimensional databases.

2.1.1 MultiDim Model

Multidim [53] is a conceptual multidimensional model which supports the representation of all multidimensional elements: dimensions, hierarchies, and measures. Hierarchies are fundamental elements in multidimensional models, since the descriptive data of the parameters under analysis are structured in the format of hierarchies. In this regard, Multidim classifies various kinds of hierarchies that exist in real world situations. A graphical formalization is also provided for the hierarchies. The graphical representation of this model is similar to Entity-Relationship model [14]. Multidim allows users to define simple, balanced, unbalanced, generalized, and few complex hierarchies using the constructs in ER model together with a set of additional constructs.

In order to give a general overview of Multidim model, an example of the model is depicted in Figure 2.1, which illustrates the conceptual schema of a sales data warehouse. In this figure, the fact relationship *Sales* is drawn by a diamond together with its measures in an attached rectangle. The fact relationship is associated to a set of levels, such as *Store*, *Time*, *Customer*, etc. These levels are organized into hierarchies by parent-child relationships, which are drawn as lines from more specific levels to more general levels. One level can participate in more than one hierarchy; thus, the notion of *analysis criterion* is defined to characterize them. For instance, the *Product* dimension in Figure 2.1 comprises two hierarchies: *Product groups* and *Distribution*. The former hierarchy contains *Product*, *Category*, and *Department* levels, while the latter contains the levels

Product and *Distributor*. Both of these hierarchies could be considered as balanced or unbalanced hierarchies, according to their schema and instance trees. An *unbalanced hierarchy* tree has only one path at the schema level. However, at the instance level some parent elements may not have associated child elements. One of the features of Multidim is to represent subtype relationships between entities as a *generalized hierarchy*. An example of such hierarchy is *Customer type* in Figure 2.1 that the *Customer* level is generalized into the *Sector* and *Profession* types of customers. Malinowski et al. give a mapping of the conceptual model to the relational and object-relational models, thus showing the feasibility of implementing Multidim in current database management systems.

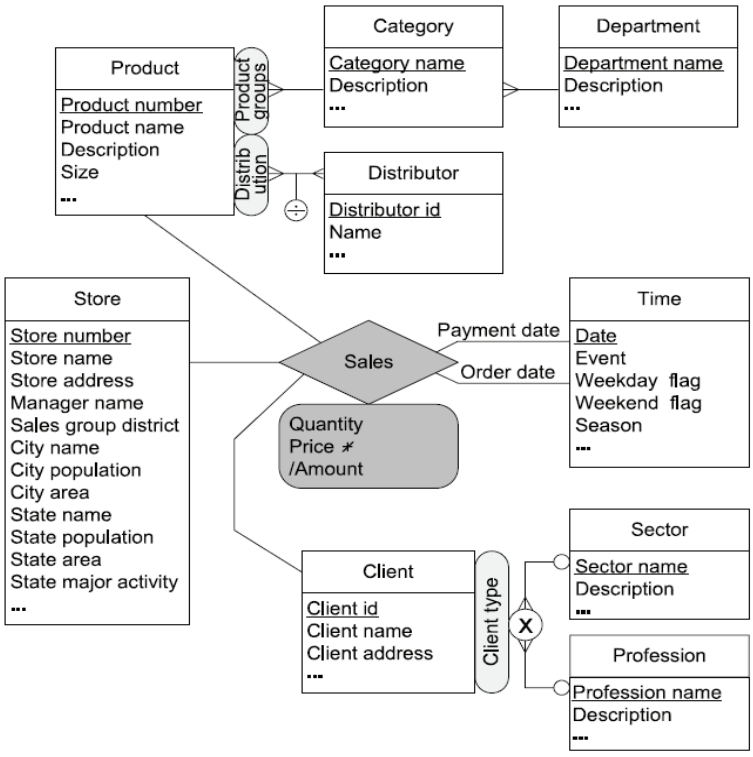

Figure 2.1: The multidim schema of a sales data warehouse [53]

2.1.2 StarER Model

StarER [73] presents a conceptual model based on a set of user modeling requirements and concepts. Specifically, the starER model integrates ER model constructs with the elements of star schemas (which are widely used in data warehouses) along with special types of relationships between the two. The set of constructs in starER includes: (1) facts, which capture events or measurements in the underlying data warehouse, (2) entities, which show objects in the environment, including those that correspond to levels in hierarchies, (3) attributes, which specify characteristics of entities, relationships or facts, and (4) relationships, which express the parent-child relations among entities and the association of entities to facts. To illustrate, consider the example in Figure 2.2, where a mortgage company is interested in keeping track of customers repayments at specific dates, for their loans on buildings. In this scenario, the event is the repayment of the loans, which is drawn by a circle as a fact set. Fact properties can be of type stock, flow, or value-per-unit. This is indicated by an "'S"', "'F"', or "'V"' on the left of the attribute illustration, respectively. Typical entity sets from the mortgage example are "'loan"' and "'day"'. Figure 2.2 shows the relationship between entity set "'loan"' and fact set "'repayment"' using entity type "'payback"'. Relationships among entity sets can be of type aggregation, specialization/generalization and membership. The specialization/generalization relationship set between customer and company or person is shown by thick arrows. In startER mode, the parent-child relationship between the two entity sets is depicted by a membership relationship. In a data warehouse, it is important to know if a membership is strict or complete. Strict membership means all instances of an entity set belong to only one instance of the higher involving entity set. For example, a "real estate" is located in only one "city". Complete membership means that all instances of an entity set belong to one higher involving entity set and that

entity set consists solely of those instances. For example, all "'branches"' belong to the "'mortgage company"' and only those "'branches"'. Therefore, the membership branch-company is strict and complete. The strictness or not of a membership relationship set is represented by cardinality of the membership and the accompanying constraint (i.e., M:1 and M¿0 means strict). Moreover, a complete membership is illustrated by a solid arrow, while a non-complete is given by a dashed arrow. In addition, a mortgage company consists of a financial department as well as an administration department, which is shown by an aggregation relationship between "'mortgage company"', "'financial dept"' and "'administr. dept"'.

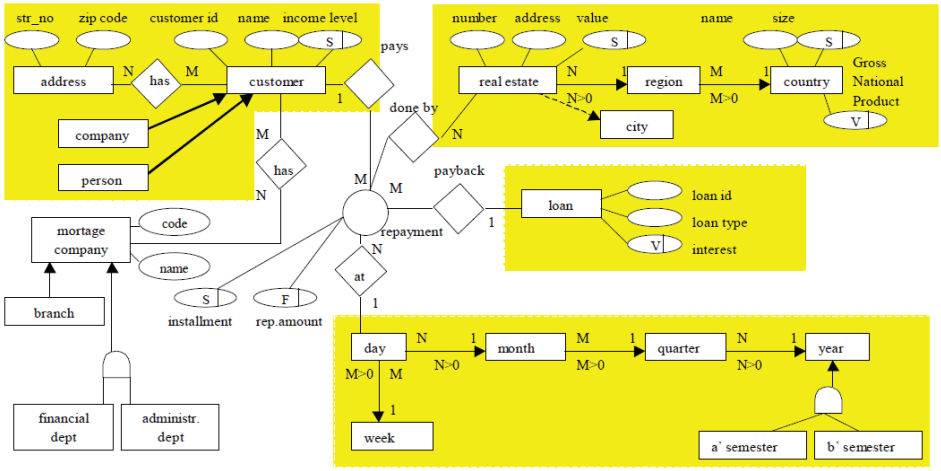

Figure 2.2: The starER model of a mortgage company data warehouse. [73]

2.1.3 Multidimensional Entity Relationship (ME/R) Model

An Entity-relationship (ER) model is a conceptual schema or semantic data model of a relational database. It was first presented by Chen [14] in 1976, and other variants of the model have been developed since. The main building blocks of this model are entity types, relationship types, and attributes. Entity is an independent existence that can be uniquely identified. Each entity could be an instance of an entity type, which

is a group. For instance, *employee* and *company* are considered as entity types. A relationship captures how two or more entities are related to one another. A relationship that associates *employee* to *company* can be referred as *works*. Entities and Relationships can both have attributes to describe them. In an ER model, *name* and *age* can be defined as attributes for an employee. Furthermore, the detailed information about the duties of an employee working in a company can be provided in *description* attribute of *works* relationship.

Regarding the multidimensional paradigm, the inherent separation of qualifying data in dimensions and quantifying data in fact tables cannot be expressed in an ER model, since all entity sets are treated equally in this model. Moreover, the semantics of the hierarchical structure of dimensions is a key feature of the multidimensional paradigm that cannot be modeled as a relationship set in ER model.

In order to support the semantics in multidimensional schemas, Multidimensional ER model (ME/R) [69] proposes a specialization of the ER model by adding new constructs suited for modeling of data warehouses. An example of a multidimensional ER model for a vehicle repairs data warehouse is depicted in Figure 2.3. By convention, a dimension level is shown as a rectangle labeled with its name. For instance, *vehicle* and *vehicle model* are levels of the multidimensional model in Figure 2.3. The arrows which are drawn from one level to another show the parent-child relationship between the two levels. This construct is called a *classification relationship* in ME/R. Figure 2.3 illustrates the categorization of vehicles into vehicle models as represented by drawing a classification relationship between *vehicle* and *vehicle model*. The cubical graphical notation labeled by *vehicle repair*, in figure 2.3 is a fact relationship. The directly connected dimension levels express the most granular data, in the related dimension, regarding the fact relationship. The attributes of the fact relationship model the measures of the fact while dimension

levels model the qualifying data.

Multidimensional Entity-Relationship introduces an easy technique for modeling of multidimensional semantics. Multiple hierarchies, alternative paths and shares dimension levels can be expressed in this model. The capability of including E/R elements in the model makes ME/R a rich conceptual model for data warehouses.

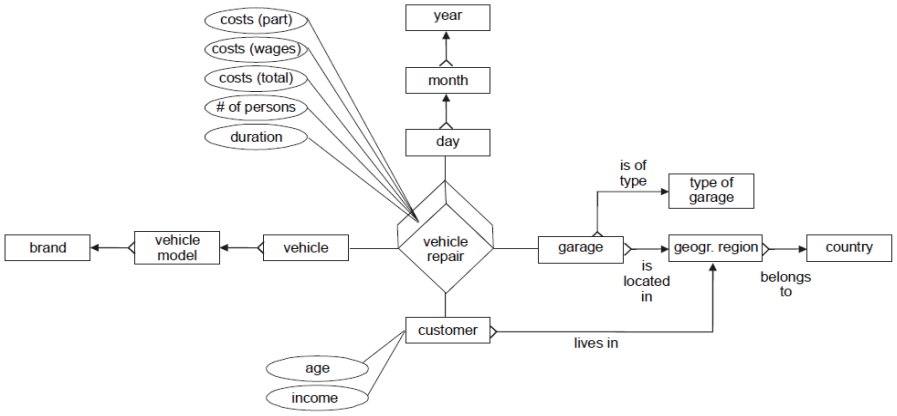

Figure 2.3: The ME/R diagram for the analysis of vehicle repairs [69]

2.2 Data Warehouse Schema Models

2.2.1 Star Schema

The **star schema** is a way to provide multi-dimensional database functionality using a relational database. It has the simplest style of data warehouse schema. The star schema comprises a fact table and its relationships to any number of dimension tables. The fact table holds the main data which are classified along differentdimension tables. Fact tables in a star schema are in third normal form (3NF) whereas dimensional tables are de-normalized. Even though fact and dimension tables are physically the same type of tables, they are different from a logical point of view. The distinguishing difference is that dimension tables have a simple primary key, while fact tables have compound

primary keys which consist of a combination of foreign keys refering to the relevant dimension primary keys. A fact table also contains numeric facts that could be at a detailed or aggregated level. Attributes of dimensions are often descriptive, textual values. Dimension tables can be joined to fact tables when relevant descriptive data is required for fact values.

One of the advantages of using the star schema is that it supports the typical commitment of organizations to relational databases. Moreover, it has simplicity from the users' point of view: no complex query is needed because joins and conditions simply involve fact tables and single-level dimension tables.

Figure 2.4 shows the relationship of the fact and dimension values within a single fact table and five dimension tables. The primary key of the fact table is composed of foreign keys: *id_d1*, *id_d2*, *id_d3*, *id_d4* and *id_d5*, each of which is the primary key of a dimension table. The non-key columns of the fact table, referred to as *measure 1* and *measure 2*, contain fact information. These columns in a dimension table are referred to as attributes, *attribute 1* and *attribute 2*, *etc.*.

2.2.2 Snowflake Schema

A snowflake schema is a variation on the star schema, in which very large dimension tables, containing hierarchies, are normalized into multiple tables. The snowflake schema is expressed by fact tables which are connected to multiple dimension tables having different levels of parent-child relationships. Such tables are frequently designed in third normal form. The decomposition of dimensions into a snowflake structure eliminates the need for joining big dimension tables when an aggregated measure is needed. As such, snowflake schema is often better with more sophisticated query tools that isolate users from the raw table structures and for environments having numerous queries with

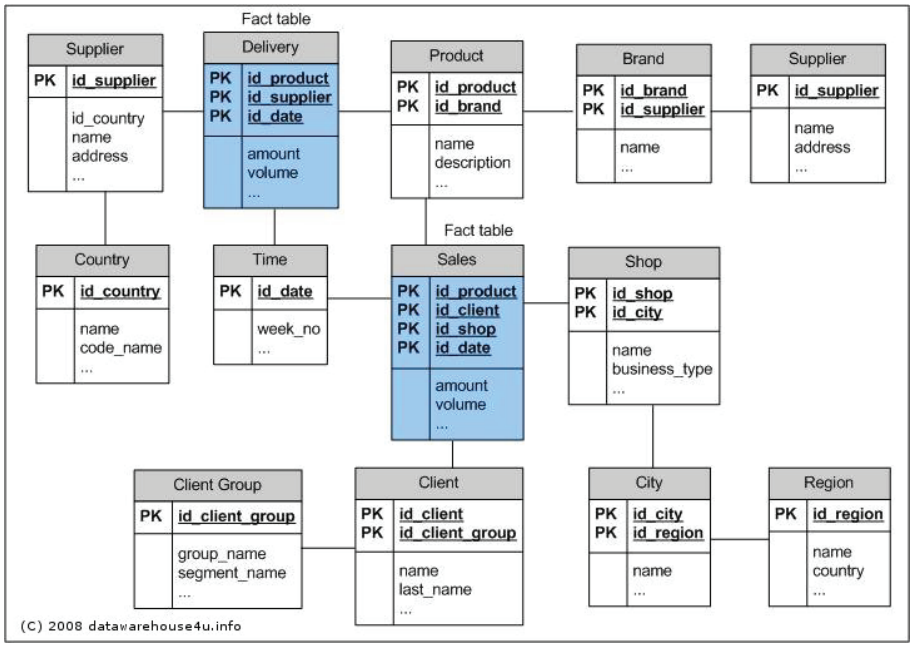

Figure 2.4: An example of star schema[1]

complex criteria.

One of the advantages of snowflake schemas is when a dimension is very sparse (i.e. most of the possible values for the dimension are null value) or in case a dimension has a very long list of attributes that may be requested in a query. In this case, designing a snowflake schema reduces the database proportion that may be occupied by dimension tables. Moreover, a snowflake schema usually reflects the way in which users think about data as it organizes data in hierarchies of parent-child relationships between dimension tables. This provides more expressive power for a snowflake schema than for a star schema.

Figure 2.5 presents a snowflake structure for the sales information of a retail company with respect to products, clients, and shops at a given date. For instance, products are categorized according to their brands and suppliers in a hierarchy. Sales information *e.g.*,volume, amount, units sold and net profit) can be aggregated according to any

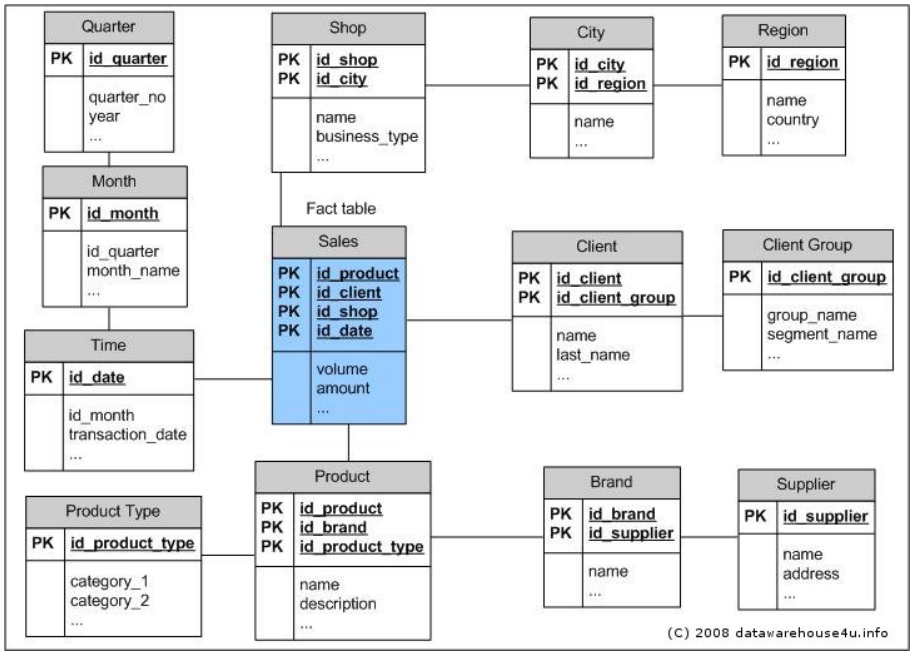

Figure 2.5: An example of snowflake schema[2]

level of granularity in any dimension *e.g.*,in the *Shop* dimension the aggregation can be computed on *Shop*, *City* and *Region*).

2.2.3 Constellation Schema

For each star schema or snowflake schema it is possible to design a fact constellation schema. This schema is more complex than a star or snowflake schema, because it contains multiple fact tables. This facilitates sharing dimension tables among many fact tables. The dimension tables can be defined as either normalized or denormalized. In a fact constellation schema, different fact tables can be explicitly assigned to the dimensions of a table, which are relevant to the given fact table. This way a fact table can be associated with a table of a given dimension whilst the other ones with less or more granular dimension tables. This may make the model difficult to manage and support, since many variants of aggregation must be considered. The fact constellation provides

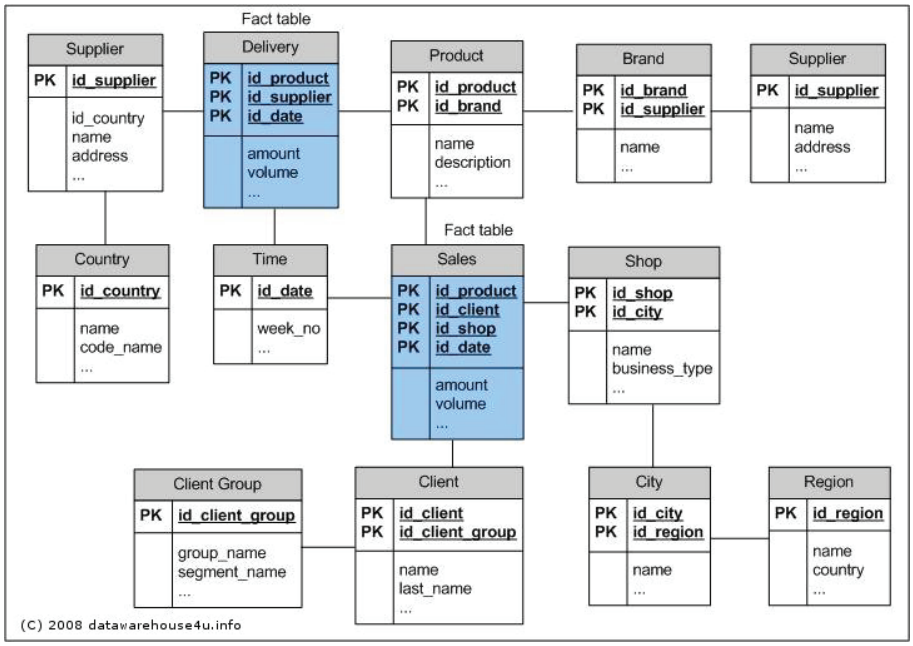

Figure 2.6: An example of constellation schema[3]

a very flexible model for data warehouses, but it can result in a much more complicated design.

An example of a constellation schema is depicted in Figure 2.6, where *Product* and *Time* dimensions are shared by *Sales* and *Delivery* fact tables. As can be seen, the fact constellation schema is a more complicated design compared to star or snowflake schemas. Moreover, since some dimension tables contain data related to more than one dimension, they are still large.

2.2.4 Data Vault

Data Vault[4] is a patent-pending technique for the next evolution in data modeling for enterprise data warehousing. The Data Vault is a detail-oriented and historical tracking set of normalized tables which follow a unique structure. It is a hybrid approach

[4]http://danlinstedt.com/about/data-vault-basics/

containing 3rd normal form (3NF) and star schema. The Data Vault is a data model that is designed specifically to meet the needs of enterprises for their data warehouses. Therefore, it is flexible, scalable, consistent and adaptable to the needs of the enterprise.

Data Vault has a minimum number of components such as the Hub, Link and Satellite Entities. A **Hub** entity is a table with a unique list of business keys that are used in everyday processes. For instance, customer number, branch number, and region number are business keys. Hubs can also contain business keys referring to the fact data of a data warehouse. **Link** entities represent a many-to-many 3NF relationship between two or more business keys. Using Hubs and Links, the data model is expressive enough to describe the flow of business information. However, it is necessary to understand the context around the keys existing in Hubs. **Satellite** entities encompass descriptive information about Hub keys. The data provided in satellites can be a set of descriptive attributes for a dimension key or fact information of a measure. The information stored in Satellite entities is subject to change over time; therefore the model should be capable of storing new or updated data at granular level.

To illustrate, an example of a Data Vault model for the data warehouse of a slow-moving vehicles sales company is shown in Figure 2.7. The *Hub Invoice* is a Hub table encompassing the invoice information from the date of invoice. The descriptive data about invoice numbers and dates are stored in the Satellite tables *Sat Invoice Amt* and *Sat Invoice Dates*. Similarly, *Sat Product Desc* describes the products that are mentioned in *Hub Product* table with their numbers. These tables contain dimension data in the data warehouse. The aggregated values in Satellite *Sat Line Item Amounts* are computed for the keys included in *Link Invoice Line Item* as the link associating the Hubs to the Satellite fact table.

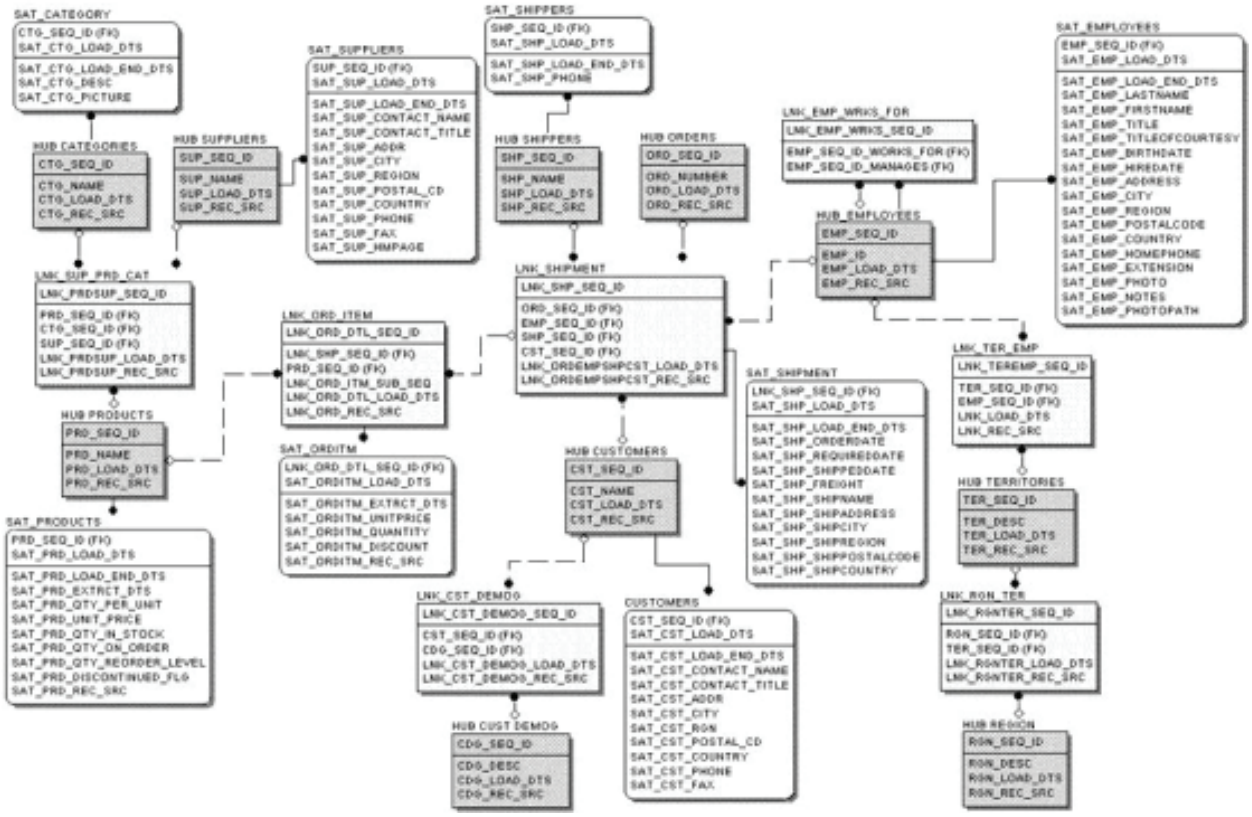

Figure 2.7: An example of datavault schema[5]

2.3 Mapping Application Objects to Databases

The main challenge in the connection of applications developed by object-oriented programming and relational databases is to make these two technologies work together seamlessly. However, the object-oriented paradigm is based on software engineering principles, while the relational paradigm is based on mathematical principles. The problem of "'impedance mismatch"' refers to technical and conceptual issues that arise when object and database artifacts need to be combined.

The connection between objects and databases remains a hard problem. Research on this problem has typically followed one of three different approaches: (1) adding database-inspired features such as transactions to programming languages (persistent programming languages), (2) extending relational databases with abstract data types, (3) creating database system toolkits and component architectures [12]. Research on persistent programming languages and applying those ideas to object-oriented languages took off in the 1980's [4]. Atkinson and Buneman took the type system and programming model of an object-oriented programming language and added features such as data persistence and atomic program executions. In the mid-1980's, the Postgres and Ingres projects, initially laid out an approach to provide storing and optimized query processing with information about the properties of data types and their operators [71, 61]. Finally, work on toolkits developed a DBMS that can be extended at almost any level, e.g. EXODUS project [13], often based on a set of kernel facilities plus tools to aid developers in building a domain-dependent DBMS.

In the context of the data translations required to bridge applications and databases, Carey and Dewitt [12] presented a survey on the connection between application objects and databases. They outlined why the first and last approaches (object-oriented databases and persistent programming languages

In fact, the ADO.NET Entity Framework [2, 59] is a platform for programming with data that defines a rich, value-based conceptual data model (the Entity Data Model) and a data manipulation language (Entity SQL) that operates on instances of this model. A middleware mapping engine supports powerful bidirectional (EDM-Relational) mapping queries and updates. This engine answers relational queries posed against the value-based conceptual layer, or against programming language-specific object abstractions.

As far as the author is aware, a few companies such as Cognos [74] have tackled the problem of translating data and data access operations between applications using multidimensional conceptual model and data warehouses. However, the translation of data to objects is performed requiring detailed information about the underlying database and multidimensional model from users.

2.3.1 Object-Relational Databases

A revolutionary approach proposed for accommodating data management needs of applications was to extend relational databases with new, user-defined data types (ADTs). It starts with the relational model and its query language, and building from there. In order to allow defining a data type in a database, it is required to define and implement its representation and functions. This type should then be registered with its size and functions within the database management system, so that it can be loaded and used in queries just like a built-in type. Among the functions provided, there should be functions to input and output the instances of the new data type. The ADT-Ingres project [61] pioneered this approach for the first time in the early '980's at UC-Berkeley.

ADT-Ingres allows a user to define abstract data types and their operations with the run-time database manager. Each ADT has an internal and external representation. The external representation is a string that represents the instance value of the ADT to

users. On the other hand, the internal representation is a user-defined data structure that represents the ADT values when it is stored and manipulated in the database system [61]. To register an ADT in Ingres, a *DEFINE ADT* command is provided and the following information must be included in the ADT specification: the name of the ADT, the maximum length of the external representation in bytes, the maximum length of the internal representation in bytes, the name of user-written C routines that covers the internal representation to external representation and vice-versa, and the name of the file that encompasses these routines. Once an ADT is defined, a user may map the columns of a table in relational database as ADT values. *ADTOP* defines a unary or binary operator on standard Ingres data types or user-defined data types. Moreover, the return value of an operator may be either kind of data types.

Each ADT operator possesses a name to be referred to within a query. ADT operators are implemented by user-defined C functions. Once an ADT operator is applied to its operands, Ingres invokes the related function and passes pointers to its operands and the buffer specified for holding the results. The user must design and implement the function such that it can cooperate with this communication protocol. With this respect, the following information should be included in the *ADTOP* command of Ingres: the operator name, the name of the user-defined C routine, the name of the file containing the C function, and the type and size of the operands and results. After defining and registering a data type, it can be used in queries and its operations can be executed on its values. For instance, a query can define the column values of a table as instances of abstract data types. Queries may also call ADT operators and retrieve ADT values that satisfy specific set of conditions.

As for other projects that build upon relational database technology, the Postgres project [71] done at UC Berkeley, the EXTRA/EXCESS effort within EXODUS project

[13] done at Wisconisn, the Starburst project [49] of IBM, and Paradise project [72] can be named. These projects, were mostly developed one decade after the emergence of object relational databases. They keep the existing basic foundations of relational databases, and explore system extensions in the context of having object types in the database.

2.3.2 Persistent Programming Languages

A different approach to addressing the perceived needs of non-traditional data-centric applications came from the programming language community. They proposed to augment existing object-oriented languages such as CLOS, CLU, or C++ with features that supported data-persistence. These features integrate database management systems into the programming language. The justification for this approach was that many applications require access to persistent data. This need can only be managed by an extension of the programming language that provides complex persistent data structures. Particularly, the applications developed with this new generation of programming languages do not face the problem of impedance mismatch, when an application meets the (relational) database system. In order to have a useful and consistent database programming language, a few object-oriented issues, such as polymorphism, object identity, the representation structure of similar values, and type inheritance, should be solved when new database types are included in the language. Research in this area was first took off, in late 80's, by Atkinson and Buneman [4] that did a survey on persistent programming languages and how data objects can enter the programming language scene.

A persistent programming language, e.g. JADE[6], natively allows objects to continue existing after the program has been closed down. In a persistent programming lan-

[6]http://www.jade.co.nz/downloads/jade/JADE_Overview.pdf

guage, a database query language is fully integrated with the host language, in the sense that they both share the same type system. Consequently, any schema alternations in databases are carried out to the program transparently. The main objective of persistent programming languages is that programmers do not need to write explicit code for fetching data into the memory or storing data in the database. This is all manipulated in a persistent programming language without any explicit coding burden. However, it is difficult to provide automatic high-level optimization in a persistent programming language. Moreover, since database modification is supported within the program, it easily happens that programming errors damage the database.

JADE is an end-to-end development platform which possesses its own programming language. The language, being a persistent programming language, integrates both a database management system and an object oriented application server. JADE allows applications to be seamlessly coded from database server to clients using the same language. However, some APIs are provided for other popular languages, such as .NET, Java, C++, that work in this framework. In terms of syntax, JADE includes necessary, but not all, features of Pascal and object-oriented languages, including C# and Java. JADE classes are structured in schemas, the same as Java packages or namespaces in .NET. However, they are different in the fact that schemas have a hierarchy, and inherit classes from super schemas. This feature is used particularly when a view definition is required in the architecture. In that case, database tables can be held as model classes in one schema, while a view can be built in a different class on top of the model classes in a subschema.

In terms of programming structure, all the code for a JADE program is stored in an object-oriented database. JADE programs are developed using a user interface that allows programmers to visually create classes and define their properties and methods.

Then, programmers may edit each method that is compiled individually as soon as it is completed. This innovative program development environment provides a number of advantages. Firstly, it allows for multi-user development, as the database maintains concurrency control. Secondly, it is possible to recode the system online as long as the parts of the system being changed are not in use. This is due to the fact that each piece of the code is a separate database object.

2.3.3 Object-oriented Databases

A radical approach towards eliminating the impedance mismatch between relational databases and applications, emerged in mid 1980's, is to combine all the features of existing databases with those of object-oriented programs, yielding an *Object-Oriented Database* (OODB) [12]. The trend in programming languages is to utilize objects, as information includes not only data but complex data types, such as video, audio, graphs, and photos. Relational database management systems are not natively capable of supporting these types of data. Thereby object-oriented databases are ideal for programmers. They can develop programs, store them as objects, and can access or modify existing objects to make new objects within the database. Migrating from relations, which were sets of tuples with simple attributes, to objects emerged the needs to work on almost all database issues in the context of object-oriented databases. The research includes query languages, indexing techniques, query optimization and processing mechanism, system architecture, user interfaces, and data model details. Three OODB projects laid the foundation in this area: Gemstone [16, 52], which was based on Smalltalk, Vbase [3], which was based on a CLU-like language, and Orion [6], which was based on CLOS.

In early 80's, Servio Logoc Corporation developed a computer system for database applications, called GemStone, in an object-oriented programming environment [16].

GemStone is an attempt to avoid the shortcomings and restrictions of commercial relational database systems. One of the problems that should be taken into account is the lack of facilities for defining new types and type operators. There are also arbitrary limits on the size of schemas and data items, e.g. fields length, the number of fields in a record, number of files, number of relations in a query, number of indexable fields, number of records in a file and depth of repeating groups. Moreover, most database systems do not support various structured objects, allowing data items as values, and modifying database schemas without database restructuring. Thus, the modeling power is limited (e.g. not supporting a hierarchy of types) and the structure of the real world might be over-simplified in modeling.

In the past, computer systems treated program variables differently from data that persisted after execution. A legitimate requirement in database systems is to access historical instances of a database. However, most database management systems lack the explicit capture of database states as part of their data models. In the context of data-centric applications, one language should be embedded in the programming language and database language. In order to avoid the problem of impedance mismatch, due to having two languages, data manipulation, systems commands, and general manipulation should be handled by a single language. The GemStone design goal was to overcome the aforementioned problems, by defining a flexible and general-purpose data model, containing a set-theoretic data models, non-procedural query language and temporal semantics. GemStone supports rich data modeling tools with a natural interface to a high-level programming language. Smalltalk-80 [25] is the basis for the language used in GemStone.

VBase [3] is an object-oriented development framework that integrates a procedural object-oriented language and persistent objects into one system. VBase encompasses

many interesting features from both object-oriented languages and databases. In fact, the most powerful aspect of this framework is that it cannot be strictly classified as a language or a database system. The system derives its heritage from CLU programming language [48] developed at MIT. Therefore, VBASE is mainly based on the abstract data type approach, rather than the object/message approach. Some of the interesting features incorporated ino VBase, from a language point of view, are: the ability to define a taxonomy of types, strong typing of objects, a block-structured schema definition language, the ability of defining constant and variables in type definition language, the ability of supporting enumeration, union, and variants, the ability to do parameterization (specifying the type of objects contained inside aggregate objects), and the ability to do method combination (base and trigger methods) and exception handling. Moreover, VBase supports most of the functionalities of a database management system, such as objects persistence, clustering objects on disk and in memory, protecting the object database from process failure, the availability of triggers and access to meta level information. VBase possesses a number of aspects that are unique to the framework. As an example, VBase provides users with the ability to customize access to objects, by using get and set functions, as well as the availability of free operations as functions not associated with a type.

ORION is a prototype object-oriented database system developed, in the late 1980's, that adds persistence to objects created and manipulated in object-oriented applications [6]. Its data model enhances a number of major concepts found in most object-oriented languages, such as objects, classes, class lattice, inheritance, and methods. ORION elaborates on three major enhancements to the conventional object-oriented data model, namely, schema evolution, composite objects, and versions. Schema evolution is the capability of making a large variety of modifications in the schema of databases without

requiring to reorganize a database. These changes vary from class definitions to the structure of the class lattice. This is provided by introducing a taxonomy for schema modifications that should be allowed in any object-oriented database system, as well as defining a framework for representing the semantics of schema modifications. Composite objects are components that recursively capture IS-PART-OF relationship between objects. A composite object hierarchy captures the IS-PART-OF relationship between a parent class and its child classes, whereby a class hierarchy expresses the IS-A relationship between a super class and its subclasses. ORION integrates these objects into an object-oriented data model in terms of schema definition, storage, retrieval, integrity enforcement, and clustering of composite objects. Furthermore, versions are variations of the same object that are stored as the history of their derivation.

2.3.4 Database System Toolkits

As we explained, different camps of researchers attack the problem of impedance mismatch in the world of objects and databases. Traditional database researchers extend the relational data model to incorporate new and complex data types. On the other hand, programming language researchers, build upon the object-oriented languages and add persistency to the application objects. A more radical approach, as discussed, is the combination of the features of relational databases to object-oriented databases and yield a new generation of database systems that incorporate the best of two worlds. In a different camp, researchers believe that a toolkit should be developed in order to aid building domain-specific database management systems. This camp advocated the belief of providing an extensible database management system with a number of kernel facilities together with tools assisting developers to create domain-dependent database management systems. Such database systems should likely have different storage structures,

different transaction handling, and even different query languages.

Known projects representing this approach are the EXODUS project [13], the DASDBS project [19], and the GENESIS project [7]. EXODUS provided a storage handler for objects as well as a persistent programming language (based on C++). The programming language (called E) was to be used for writing new access methods and query operators; EXODUS also offered a query optimizer generator [13]. More than GENESIS features, DASDBS also provided a complex object storage manager with a novel, multi-layered kernel containing transaction functionalities. Developers may use these facilities to build a domain-dependent data model and query layer. As another example of database toolkits, GENESIS consisted of a set of composable storage and indexing components plus a "'database system compiler"' for assembling an appropriate storage manager from a specification provided by a developer.

2.3.5 New Generation of solutions for impedance mismatch problem

Among the mentioned approaches, not all of them were able to make it to the state of art of this area. Carey and Dewitt [12] discuss why object-oriented databases and persistent programming languages were not more successful with the research community and developers. They predicted that object-relational databases would antedate other approaches in 2006. In fact, many recent database systems, namely DB2 UDB V5.2 [11] and Oracle [42], possess an object layer that uses a hardwired object-relational mapping on top of a conventional relational system. DB2 UDB V5.2 offers a new object-relational data definition language (DDL) and data manipulation language (DML) in order to support hierarchies and user-defined structured types, table of objects, object views, semantic extension of SELECT, UPDATE, DELETE and INSERT statements

in order to support operations on hierarchies of tables and views [11]. Oracle [42] is another example of an object-relational system that provides a type system, an object storage, an object cache, a query and indexing framework, server-based Java virtual machine, as well as support for multimedia data types. However, the features provided by current object-relational databases are only used for storing multimedia and spatial data types, and not all types of enterprise data [26]. This is due to the fact that the cost of transferring legacy databases is high; object-relational databases do not provide enough programming language integration.

In a modern data-centric application that manipulate different data representations, such as tables and rows in databases and objects in a programming language, there constantly exists the problem of translating data and data access operations. As an example, in object-relational databases, the instances of an object may be scattered across several relational tables. A common solution of isolating business logic from data manipulation is to use a *data access layer* that handles necessary transformations among objects and tables. In this approach, the features and components of databases and programming languages are not integrated together; instead, the data access layer allows these components to talk to each other. In fact, a data access layer provides an *updatable client-side view* that expresses persistent data as business entities (or programming language objects) [58] or sometimes vice versa. The core of a domain-independent data access layer is a *mapping set* that establishes a relationship between different data models (business objects and tables). Using mappings, the process of generating views can be manipulated automatically, instead of hardwiring the data access part for a specific application. Therefore, *mapping driven data access*, the challenge of translating data and data access operations, is an approach for solving the constant problem of impedance mismatch.

Mapping driven data access follows two major lines of research: (1) filling the gap

between application objects and database tables by defining views, (2) supporting updates through mappings. The updatability requirement is difficult, since it deals with the consistency of data across mappings. Moreover, Dayal and Bernstein [18] showed that finding a unique sound update translation for views, is not simply possible, because of the intrinsic challenge in the update behaviour through views. Furthermore, in a mapping-driven application, update translations might go beyond one single view. In other words, the access layer can require multiple view operations in order to preserve the consistency of underlying data. As an example of recent research in this area, Bohannon et al. [10] and Foster et al. [23] work can be mentioned. They developed a bidirectional mechanism named "'lenses"' that uses *get* and *putback* functions. However, as explained in Section 1, in the scope of this thesis, we do not deal with satisfying updatability need in data warehouses. Thus, in the rest of this section, we provide an overview of existing projects that are developed for encapsulating the need of views between business objects and database tables.

2.3.6 Clio

Clio [22] project addresses the problem of information integration by defining the concept of *non-procedural schema mappings* to express the relationship between two heterogeneous schemas. The relationship between elements of schemas are described by *correspondences*, which can be specified by a schema matcher or, alternatively, a data designer or expert. The correspondences are imaginary lines between schema elements that should contain related data. Figure 2.8 illustrates two schemas in a mapping scenario. The source schema, shown in the left, contains three relations whose attributes are mapped to a nested target schema, shown in the right, through a set of matchings and correspondences. The line v_1 states that what is called a *company name* in the

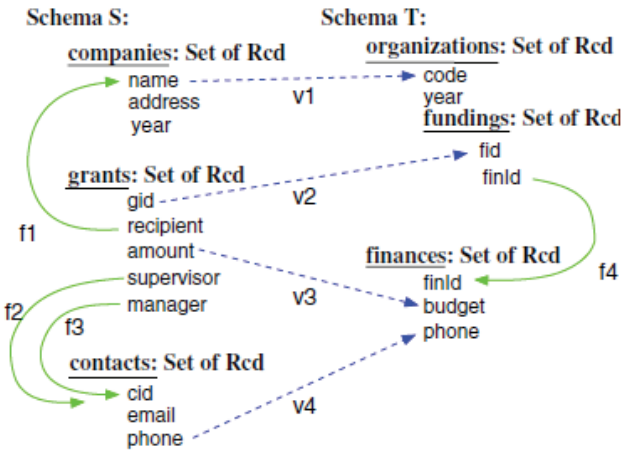

Figure 2.8: A source and a target schema in a clio mapping scenario [22]

source schema is referred to as an *organization code* in the target schema. Although two schemas have *year* element, no arrow has been specified between them, since the data administrator or matching tool believes that these elements do not populate the same concept. The arrows drawn between attribute elements of the same schema represent referential constraints provided as part of the schema definition. For instance, the curved line f_1 in Figure 2.8 is a foreign key or an inclusion dependency, indicating that all values of *grants.recipient* should appear in *companies.name*. The schemas may also be XML schemas containing referential constraints in the form of *keyref* definition (for example f_4). Clio exploits source and target schemas together with correspondences to generate a set of alternative mappings.

In the Clio project, correspondences can be interpreted as the connection between values in the source schema that have been specified to populate a target schema element. This interpretation can be formally represented by a source-to-target *tuple generating dependency* (tgd) [8]. A tgd for the correspondence v_1 is as follows:

$$\forall\ \underline{n}\text{,d,y } \textbf{companies}(\underline{n}\text{,d,y}) \rightarrow \exists\ \text{y}'\text{,F } \textbf{organizations}(\underline{n}\text{, y}'\text{, F})$$

This is a simple uni-directional mapping which describes that there must be an *organizations* tuple for each *companies* tuples, which carries the same *code* as the company's *name*. The variable n represents the correspondence between the attributes of elements.

The correspondences do not retain any information about the semantic relation of data values inside the target or source schemas. For instance, as shown in target schema of Figure 2.8, the *fundings* element is nested inside the *organizations* element. Also, in the source schema, the attributes of *grants* element refer to *companies* and *contacts* elements. The objective of Clio was to extract the hidden semantics among schemas, according to the relation of elements in a single schema and inter-schemas correspondences. In Figure 2.8, *fundings* is nested in *organizations* in the target schema, additionally, *grants* element has a referential constraint to *companies* element. The correspondence between *grants* and *fundings* elements demonstrates a real-world connection from the *association* between *fundings* and *organizations* to that of *grants* and *companies*. In general, there may be many ways of associating the elements within a schema. Clio uses chase [51] technique, a logical inference mechanism, to find these association paths by referential constraints and nested structures.

Mappings can be written such that they reflect the associations of elements within each schema plus the correspondences of schema elements. A mapping expressing the respective association between *company names* and *grant gids* in the source to organizations and fundings is shown in the following association:

$$\forall\ \underline{n},\mathrm{d},\mathrm{y},\underline{g},\mathrm{a},\mathrm{s},\mathrm{m}\ \textbf{companies}(\underline{n},\mathrm{d},\mathrm{y}),\ \textbf{grants}(\underline{g},\underline{n},\mathrm{a},\mathrm{s},\mathrm{m}) \rightarrow$$
$$\exists\ \mathrm{y}',\mathrm{F},\mathrm{f}\ \mathrm{organizations}(\underline{n},\ \mathrm{y}',\ \mathrm{F}),\ \mathrm{F}(\underline{g},\mathrm{f})$$

The similarity of variable n in *companies* and *grants* describe the referential constraint among these two elements; the same case for *companies* and *organizations* which declare the correspondence v_1.

Clio generates a set of mappings according to relations between schema element associations, specified by correspondences. These mappings are a form of sound GLAV (global-and-local-as-views),that assert a relationship between a query over the source and a query over the target. In Particular, Clio uses sound mappings, where there is a containment relationship between queries. Typically, as in data integration, the result of the source query in a mapping is contained in the target query. This implies the freedom of mapping multiple source elements to a single target element, since such mappings do not restrict what data can be in the target.

The Clio mapping generation algorithm extracts associations in each schema applying the chase technique [51] and following integrity constraints and nested structures of schemas. All pairs of different schema associations are considered by the algorithm. However, not all of them generate actual mappings. Some of them are not connected through correspondences and are discarded. In addition, some pairs cover other associations, thus should be discarded. Intuitively, those associations that cover the same correspondences and are not subsumed by already selected associations are considered as candidates for creating mappings. Clio uses a specific language to represent mappings. An example of a complex mapping generated for the schemas in Figure 2.8 is as follows:

foreach c *in* **companies**, g *in* **grants**
  where c.**name** = g.**recipient**
exists o in **organizations**, f *in* **fundings**, f′ *in* **finances**
  where f.**finId** = f′.**finId**

with c.**name**=o.**code** *and* g.**gid**=f.**fid** *and* g.**amount**=f′.**budget**

This mapping is generated following the steps described for the Clio algorithm. An association is found between *grants* and *companies* following the referential constraints in the target schema of Figure 2.8. Another association can be found for *organizations*, *fundings* and *finances* elements in the source schema. The connection between schema elements in each association is indicated by the conditions in the *where* clauses of the mapping. For instance, the referential constraint between *fundings* and *finnaces* elements is indicated by $f.finId = f'.finId$. Moreover, user-defined correspondences between schemas are translated into conditions in *with* clause of the mapping. These associations cover correspondences v_1, v_2 and v_3. The above mapping is created using these three correspondences.

2.3.7 Object and relational mapping (ORM) with ADO.NET

Miscrosoft *ADO.NET Entity* [2], a part of .NET framework, is a platform for object-oriented programming against data in relational databases. It allows data-centric applications and services to act at a higher level of abstraction than tables via a new *Entity Data Model* (EDM) and mappings between conceptual schemas and database schemas. The Entity Data Model integrates the concepts of $Entity - Relationship$ (ER) model to classic relational model. The building blocks in the EDM are entities and associations. As in ER model, entities represent identified objects, while associations describe the relationship of two or more entities. EDM is a value-based model as in relational models; it can be attached to multiple database management systems. Similarly, due to having entity constructs, EDM can be layered by object oriented programming languages on top; the constructs of EDM are accessed by objects in the programming language.

Entity SQL is a derivative of SQL designed for querying Entity Data Models. It provides constructors, member accessors, type interrogation, and relationship navigation in order to support EDM constructs. It also groups EDM constructs for types and functions using namespaces. The EDM and Entity SQL represent a rich data model and data manipulation language for a data layer and intended to model and manipulate data at a level of structure and semantics that is closer to the needs of reporting and data-centric applications.

In ADO.NET, the problem of impedance mismatch between the Entity Data Model as a conceptual model and relational model of databases is addressed by defining a set of declarative mappings [59]. The mappings are automatically translated into bidirectional views. In contrast to the unidirectional mappings generated in Clio [22], a set of views can be leveraged for supporting updates in an elegant way [2]. Two types of views are required: views that are used to answer queries and views that allow us to do database update maintenance through views. These views are expressed in Entity SQL. *Query views* describe entities in terms of relational tables, while *update views* express tables in terms of EDM entities. Melnik et. al, outline an approach for generating query and update views under *roundtripping* condition [59]. *Roundtripping criterion* guarantees that all entity data can be persisted and reassembled from the database in a lossless fashion. This approach has been implemented and released in the commercial product. Their approach is based on answering-queries-using-views techniques for exact rewritings [27].

To illustrate, a mapping scenario is depicted in Figure 2.9. An XML representation of the mapping is also shown in Figure 2.10. The conceptual side describes *order* and *salesperson* entities via the entity type definitions for *ESalesPerson*, *ESalesOrder* (and the *EStoreSalesOrder* subtype). The *ESalesPersonOrder* association represents a 0..1 :

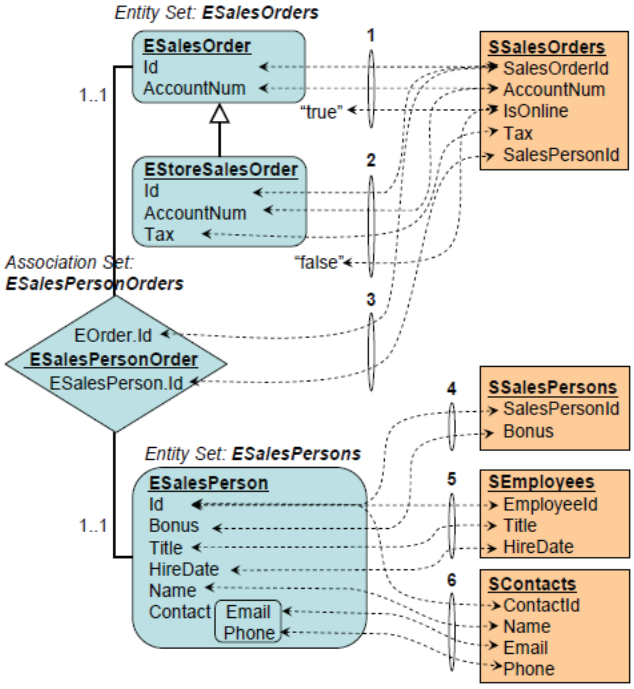

Figure 2.9: Mapping between an entity schema (left) and a database schema (right) [2]

N association relationship between *ESalesPerson* and *ESalesOrder*. On the store side, there are four tables *SSalesOrders*, *SSalesPersons*, *SEmployees*, and *SContacts*. The declarative mapping between these two schemas is described in terms of queries on the entity schema and the relational schema as shown in Figure 2.11. The first fragment explains that the set of (Id, AccountNum) values for all entities of type *ESalesOrder* in *ESalesOrders* is the same as the set of (SalesOrderId, AccountNum) values retrieved from the *SSalesOrders* table for which *IsOnline* column is true. The second fragment is similar to the first one. The third fragment maps the association set *ESalesPersonOrders* to the *SSalesOrders* table and says that each association entry corresponds to the primary key, foreign key pair for each row in this table. The last three fragments say that the entities in the *ESalesPersons* entity set are split across three tables *SSalesPersons*, *SContacts*, *SEmployees*.

At compile time, these mappings are compiled into bidirectional views, expressed in Entity SQL, that can be used in answering queries during the runtime. Figure 2.12 describes the query and update views generated by the mapping compiler for the mappings provided in Figure 2.11. In fact, these views are more complex than the input mappings, since they explicitly specify the data transformation for retrieving or updating data. For instance, in order to reassemble the *ESalesPersons* entity set from the relational tables, it is required to perform a join between *SSalesPersons*, *SEmployees*, and *SContacts* tables, as shown in (*QV3*). In addition, as in *QV1 ESalesOrders* entity set is built from the *SSalesOrders* table so that either an *ESalesOrder* or an *EStoreSalesOrder* is instantiated depending on whether or not the *IsOnline* flag is true.

Manually creating query and update views that satisfy the roundtripping criterion requires high level of database expertise; therefore, the current ADO.Net Entity framwork provides a module for producing a set of complete views automatically. Here, we describe the view generation approach of ADO.NET in general and discuss that it satisfies the data roundtripping problem.

Assume that P is a schema consisting of a set of relations (partitions) plus a set of schema constraints σ_P. P is called a *partitioned schema* for S, relative to a query language L, if there is a procedure that rewrites each query $g \in L$ on S using a unique set of partitions in P. Each f such rewritings should be injective on the subschema P_{exp} made by the partitions used in the rewriting and the respective schema constraints from σ_P. Furthermore, the rewriting of the identity query on S uses all partitions in P. According to the last two conditions, two bijective views can be found that satisfy the equivalence of P and S. As shown in Figure 2.13, query g partitions S into $P \subseteq P_{exp} \times P_{unexp}$ such that $p_{exp} : S \rightarrow P_{exp}$ and $p_{unexp} : S \rightarrow P_{unexp}$ are view complements [5] that together give P. Considering the condition of having bijective rewritings on subschema P_{exp}, g

```
<?xml version="1.0" encoding="utf-8"?>
<Schema Namespace="AdventureWorks" Alias="Self" …>
  <EntityContainer Name="AdventureWorksContainer">
    <EntitySet Name="ESalesOrders"
               EntityType="Self.ESalesOrder" />
    <EntitySet Name="ESalesPersons"
               EntityType="Self.ESalesPerson" />
    <AssociationSet Name="ESalesPersonOrders"
           Association="Self.ESalesPersonOrder">
      <End Role="ESalesPerson"
           EntitySet="ESalesPersons" />
      <End Role="EOrder" EntitySet="ESalesOrders" />
    </AssociationSet>
  </EntityContainer>

  <!-- Sales Order Type Hierarchy-->
  <EntityType Name="ESalesOrder" Key="Id">
    <Property Name="Id" Type="Int32"
              Nullable="false" />
    <Property Name="AccountNum" Type="String"
              MaxLength="15" />
  </EntityType>
  <EntityType Name="EStoreSalesOrder"
              BaseType="Self.ESalesOrder">
    <Property Name="Tax" Type="Decimal"
              Precision="28" Scale="4" />
  </EntityType>

  <!-- Person EntityType -->
  <EntityType Name="ESalesPerson" Key="Id">
    <!-- Properties from SSalesPersons table-->
    <Property Name="Id" Type="Int32"
              Nullable="false" />
    <Property Name="Bonus" Type="Decimal"
              Precision="28" Scale="4" />
    <!-- Properties from SEmployees table-->
    <Property Name="Title" Type="String"
              MaxLength="50" />
    <Property Name="HireDate" Type="DateTime" />
    <!-- Properties from the SContacts table-->
    <Property Name="Name" Type="String"
              MaxLength="50" />
    <Property Name="Contact" Type="Self.ContactInfo"
              Nullable="false" />
  </EntityType>
  <ComplexType Name="ContactInfo">
    <Property Name="Email" Type="String"
              MaxLength="50" />
    <Property Name="Phone" Type="String"
              MaxLength="25" />
  </ComplexType>
  <Association Name="ESalesPersonOrder">
    <End Role="EOrder" Type="Self.ESalesOrder"
         Multiplicity="*" />
    <End Role="ESalesPerson" Multiplicity="1"
         Type="Self.ESalesPerson" />
  </Association>
</Schema>
```

Figure 2.10: EDM schema of Figure 2.9 in XML format [2]

SELECT o.Id, o.AccountNum FROM **ESalesOrders** o WHERE o IS OF (ONLY ESalesOrder)	=	SELECT SalesOrderId, AccountNum FROM **SSalesOrders** WHERE IsOnline = "true"
SELECT o.Id, o.AccountNum, o.Tax FROM **ESalesOrders** o WHERE o IS OF EStoreSalesOrder	=	SELECT SalesOrderId, AccountNum, Tax FROM **SSalesOrders** WHERE IsOnline = "false"
SELECT o.EOrder.Id, o.ESalesPerson.Id FROM **ESalesPersonOrders** o	=	SELECT SalesOrderId, SalesPersonId FROM **SSalesOrders**
SELECT p.Id, p.Bonus FROM **ESalesPersons** p	=	SELECT SalesPersonId, Bonus FROM **SSalesPersons**
SELECT p.Id, p.Title, p.HireDate FROM **ESalesPersons** p	=	SELECT EmployeeId, Title, HireDate FROM **SEmployees**
SELECT p.Id, p.Name, p.Contact.Email, p.Contact.Phone FROM **ESalesPersons** p	=	SELECT ContactId, Name, Email, Phone FROM **SContacts**

Figure 2.11: Representation of the mapping arrows in Figure 2.9 as pairs of queries [2]

can be represented as $h \circ p_{exp}$, where h is an injective function. Assuming that h' is an exact rewriting of the identity query on P_{exp} using h, the function h' can reconstruct P_{exp} from V. As the next step, the update view u and merge view m can be reconstructed as follows:

$$u := foh' \circ r[., 0]$$
$$m(s_1, s_2) := r(p_{exp}(s_1), p_{unexp}(s_2)).$$

where 0 is the state of P_{unexp} where all relations are empty, the view $r[., 0]$ is such that $r[., 0](x) = y$ iff $r(x, 0) = y$, and $s_1, s_2 \in S$. Assuming that f is injective and *Range(f)* $\subseteq$ *Range(g)*, it is easy to show that u and m meet the roundtripping criterion. This criterion holds if we choose the view r in such a way that

$$u := foh\prime \circ r[., 0]$$
$$m(s_1, s_2) := r(p_{exp}(s_1), p_{unexp}(s_2)),$$

One of the ways to obtain such r is to left-outer-join exposed partitions with unexposed partitions that agree on keys. Melnik et al. [59], proved that it is always possible to

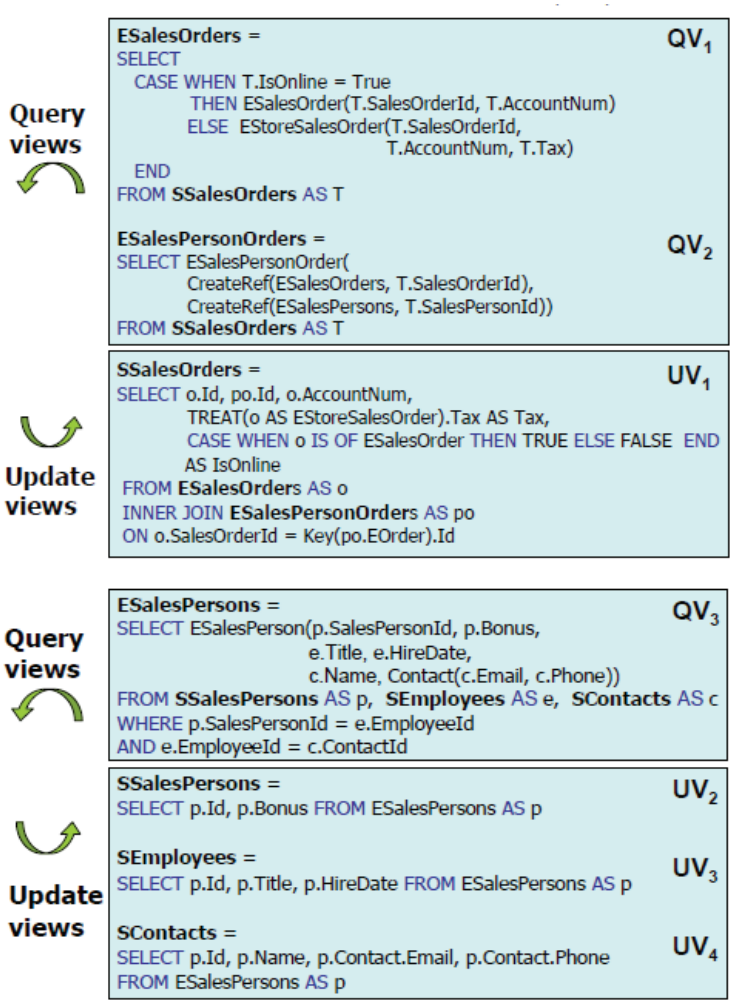

Figure 2.12: Bidirectional views (query and update views) for mappings in Figure 2.11 [2]

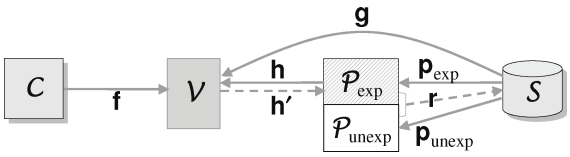

Figure 2.13: Partitioning idea in generating views [59]

create the views h, h', p_{exp}, p_{unexp}, and r that satisfy the conditions just mentioned. This proof implies that the mapping compilation algorithm is complete and it allows generating query, update, and merge views for each given valid mapping. The function g is injective if and only if P_{unexp} is empty, meaning, it has zero partitions. This property can be checked the injectivity of f in the previous construction takes the place of g, and apply the partitioning scheme to the conceptual schema C in a symmetric fashion.

In brief, the mapping compilation uses a divide-and-conquer method to subdivide the mapping into independent sets of fragments. Two mapping fragments are called *dependent* if their fragment queries share a common symbol or some integrity constraint spans their symbols; in this case, they are categorized into the same subset of fragments. Then, schema S and g are partitioned; g is written as $h \circ p_{exp}$. The function h' is produced as an exact rewriting of identity query on P_{exp}. As a result, u and m functions can be obtained and simplified. The same process should be performed for C and f. In this process, f' should be produced as an exact rewriting of the identity query on C. The query view q is then constructed and simplified as $f' \circ g$.

2.3.8 Object and relational mapping (ORM) with Hibernate

Hibernate [47] is a framework provided in the Java programming language, for mapping an object-oriented model to relational databases. Hibernate is released under the Lesser GNU Public License (LGPL), which is sufficient for use in open source and commercial applications. Its main feature is mapping from Java classes and data types to database

tables and SQL data types. Hibernate also facilitates data query and retrieval and eliminates the need for manual, hand-tailored data processing. In Hibernate, Java classes can be mapped to database tables through the configuration and XML files or using Java Annotation. An *annotation* is a syntactic metadata that can be added to Java classes, methods, variables, parameters and packages. When the mappings are introduced in an XML file, Hibernate translates the file into necessary annotations in the related classes.

The XML mapping documents define the object model and generate database table and constraint creation scripts. Various inheritance mapping strategies and all entity association styles (one-to-many, one-to-one, and many-to-many) are supported in the Hibernate framework. It also supports bidirectional and unidirectional associations as well as fine-grained composition for dependent value objects. Hibernate applies the mappings in order to retrieve data from databases and copy it to a class. In the opposite direction, created objects within the program can be saved to the database, by being transformed to one or more tables. In Hibernate terminology, retrieving and storing operations on objects and tables is called 'object relational mapping'. Furthermore, the process of saving data to a storage is called 'persistence'. Hibernate includes a Core API for application code as well as an Extension API for customizations. It also provides a Metadata API for applications that request for access to persistence metadata to manage runtime mapping creation and revision. *Hibernate Query Language* (HQL) is a SQL-like language that is written against Hibernate's data objects. Queries may also be expressed in the native SQL. An object-oriented alternative to HQL is *Crieteria Queries* which is also provided by Hibenrate.

2.3.9 State of Art of Decision Making Applications and Multi-dimensional databases

Business Intelligence (BI) refers to the proposal of a computer-based technique used in intelligent exploration, integration, aggregation and multidimensional analysis of data originating from various information resources [50]. BI standard systems integrate data of various information systems of an organization, coming from various environments *e.g.*, statistics, financial and investment databases. Business Intelligence is Decision Support System 'DSS', aiming to support better business decision-making [65]. Therefore, BI technologies provide services in reporting, online analytical processing, data mining, text mining, analytics, business performance management, benchmarking and predictive analytics. The main objective of using BI tools in an organization is to contribute to the improvement and transparency of information flow and knowledge management. This enables organizations to follow profitability of their products and effectiveness of their processes, analyze expenditures, monitor corporate environments and discover business anomalies and frauds.

Forrester Research[7] defines BI as: 'Business Intelligence is a set of methodologies, processes, architectures, and technologies that transform raw data into meaningful and useful information used to enable more effective strategic, tactical, and operational insights and decision-making'. This definition declares how business intelligence and data warehouse are related in spite of being separate in some domains. According to the Forrester definition, business intelligence includes technologies such as data quality, data integration, master data management, text and content analysis, and any topic that is related to Information Management. Therefore, data preparation and usage can be considered as segments of business intelligence. As a result, data warehouse management is

[7]http://www.forrester.com/rb/research

an important topic in business intelligence applications. In this section we briefly discuss a few existing business intellgence frameworks that can be used in any environment.

2.3.10 Cognos 8

Cognos 8 [74] is a web-based software product designed to address the challenges of reporting and analysis on an enterprise scale. The architecture of Cognos 8 was designed such that it provides scalability, openness, and availability. For this reason, it uses a number of platform-independent technologies such as Extensible Markup Language (XML), Web Services Definition Language (WSDL), and Simple Object Access Protocol (SOAP). As a result, Cognos 8 can be integrated with any existing infrastructure on various platforms.

The *Data Manager* of Cognos 8 takes care of data integration. It helps extracting data from source systems and data files, transform the data, and load it into a data warehouse, data mart, or report area. *Framework Manager* is the Cognos 8 modeling tool for creating and managing business-driven metadata of data in use. Metadata may be published for use by enterprise applications or reporting/analysis tools of Cognos 8. OLAP cubes are designed to retain at least metadata for business intelligence and analytical reporting. These cubes are published and accessed by dashboards and web-based reporting tools. The specification and metadata of reports provided by Cognos 8 are in XML format.

Since cube metadata may change as a cube is developed, Framework Manager uses the minimum amount of information needed to model a cube. The necessary components are cube dimensions, hierarchies, levels and measures. These components are objects that should be mapped to data elements in databases. The mappings to database tables are defined by *Query Subjects*. They are the basic building blocks in Framework Manager.

A *Query Subject* is a select query, written by the developer of the model, that pulls out a set of needed columns from the database [74]. Comparing to the architecture of Clio [22], query subjects are mappings that are returned as the output of Clio's schema mapping generation algorithm; the inputs to the algorithm are the schema of underlying data sources as source schema, the conceptual model defined in Framework Manager as target schema, and correspondences between levels defined in Framework Manager model and columns of tables in source databases. The main structural distinction between the Clio project and a Cognos application is that the mappings (query subjects) in Cognos applicaion should be defined manually by developers; whilst, Clio has a separate mechanism to extract mappings automatically. Moreover, the framework developed in Cognos 8 specifically supports multidimensional modeling of data warehouse. Nonetheless, the target and source schemas in Clio are not specifically adjusted for multidimensional schemas. Therefore, a framework which is designed exclusively for modeling data warehouses and generates mapping views automatically is considered as the next generation of Business Intelligence tools.

2.3.11 Conceptual Integration Model (CIM)

In a data-centric application, users describe the properties and formats of the business information they need, and the system satisfies this request. Achieving this requires raising the level of abstraction, without losing the connection to the real data. The *Conceptual Integration Model* (CIM) framework [67] allows users to specify their data access needs by a conceptual model, which is associated to a multidimensional database. The conceptual model, provided by CIM, is called *Conceptual Visual Language* (CVL). It is an extension of Entity-Relationship Modeling [14], borrowing concepts from StarER [73] and MuliDim [53] models, discussed in Section 2.1.2 and Section 2.1.1. The *Store Visual*

Language (SVL) describes the (relational) multidimensional model of the underlying data warehouse. In order to translate the conceptual model CVL to store model SVL, a visual mapping language is created and called as *Mapping Visual Language* (MVL). There is a second representation for each of conceptual, store and mapping models, which is based on XML format.

Figure 2.14 shows a small fragment of the Dream Home data warehouse [15] modeled in the Conceptual Integration Model. Dream Home is a property management company with branch offices in cities of North America. The company makes analytical decisions based on a few numerical measures such as *staff commission* and *revenue* obtained from renting/selling a property in a specific branch. The CVL of the conceptual model is shown on the left hand side of Figure 2.14, and a physical model (SVL) of the Dream Home data warehouse is depicted on the right-hand side. The SVL captures the referential integrity constraints of the store model, by intra-model dashed arrows.

In the conceptual model of Figure 2.14, *Branch Location* and *Client* (shadowed rectangles) are dimensions representing measures in the *Sale* fact table (shadowed diamond). Non-shadowed rectangles (e.g., *Branch*, *City*, and *Client*) express levels in the dimensions. These levels are organized into hierarchies through parent-child relationships, which are drawn as arrows from levels with more specific data to levels containing more general data. The hierarchy in *Branch Location* specifies that all branches roll up to *City*, *Country* and *Region*. However, some cities roll up to *Province* (e.g., in Canada), some roll up to *State* (e.g., in the USA), and yet others roll up directly to *Country* (e.g., the case of *Washington DC*).

Many existing models such as MDX [70], Universes [29], Mondrian [1], and Framework Manager [74] could have been explored as a conceptual modeling language for multidimensional modeling instead of the conceptual models (i.e., CDL and CVL) pre-

sented in CIM. However, there are two reasons that can be mentioned about not using the aforementioned models for the purposes of creating top-down Business Intelligence tools. First, the designers of any Business Intelligence tools are interested in providing a user-oriented, lightweight modeling language. None of the above candidates display an ease of use. For instance, one can hardly imagine an average executive who would be an expert in MDX, which is a highly technical language whose knowledge requires deep expertise in multidimensional modeling. Additionally, the other candidates present a level of complexity that puts them at a level of abstraction that is midway between the traditional multidimensional schemas and CIM framework. Second, the constructs of a modeling language should be easily assigned a clear and declarative semantics with respect to their persistence nature. The semantics of components of the CIM can be expressed using Datalog rules, which are not transparent to users [67]. None of the other candidates displays a clearly defined semantics that serves as its formal foundation.

Although CVL and SVL are expressive models for representing conceptual and store sides, the problem of impedance mismatch emerges when an application needs to access the data with the SVL model through objects that refer to conceptual model constructs. The gap between the CVL (conceptual) and the SVL (physical) models is filled by defining correspondences between attributes of entities in the different models. To illustrate, the SVL shows that cities are physically stored in two different tables, *SCity* and *STown*. The attribute mapping between these tables and the CVL *City* is represented by the arrows linking the *cityID* and *name* attributes in the three entities. These user-defined correspondences, which can also include value conditions, are part of the *Mapping Visual Language* (MVL). As can be seen, mappings provided in MVL are represented at the attribute level. Therefore, they only provide information about the association of attributes of the constructs in CVL to table columns expressed in SVL, regardless of

the meaningful structure of constructs in the conceptual model and tables in the data warehouse. As a result, the CIM framework is a promising starting point for a new generation of Business Intelligence tools.

The Conceptual Integration Model framework is an ongoing project which subsumes the objective of this thesis. At design time, a business user uses the capabilities of CVL and SVL to specify what business information is required. Then, she expresses the mapping of these models in an MVL representation. At compile time, a system will transform the user conceptual model together with the mappings, which are at the attribute granularity level, into more complex views over the logical multidimensional representation. Finally, at run time, the query answering engine creates requested reports asked against the CVL.

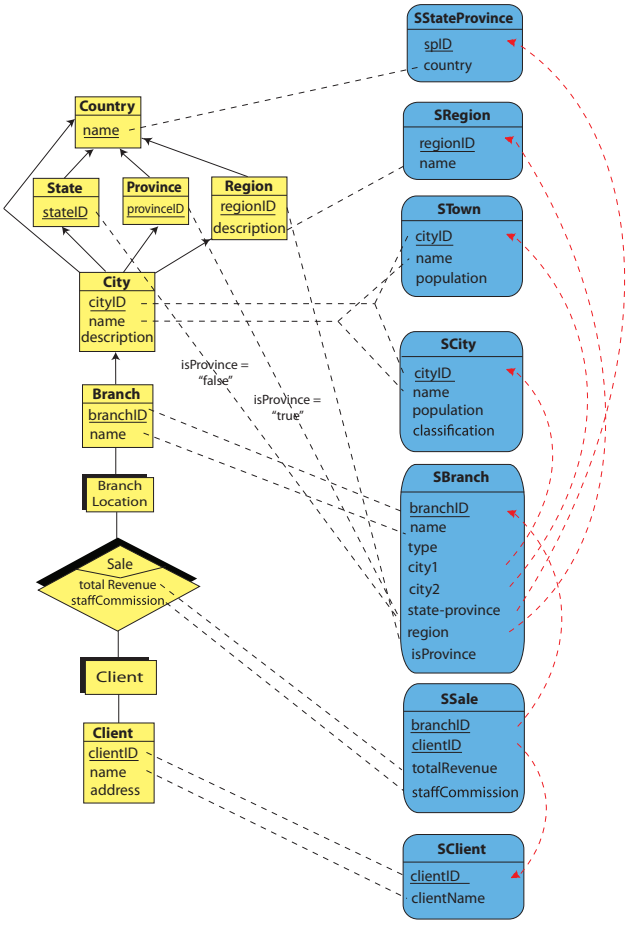

Figure 2.14: Dream Home CIM Visual Model: CVL (left), SVL (right) and MVL (left-right dashed lines).

Chapter 3

Conceptual Integration Data Model

In this chapter, we develop an approach to mapping that is grounded in conceptual modeling using the Conceptual Integration Model (CIM). First, a scenario is discussed in the context of CIM framework. Next, its graphical conceptual model, known as Conceptual Visual Language(CVL), is represented in a logical format. The properties of the conceptual model are formally expressed and discussed. The graphical and logical representation of CIM has an XML representation which is called Conceptual Description Language (CDL). The Store Visual Language(SVL) graphically models the relational schema of the data warehouse, where the data is physically located. The corresponding XML format of store model in CIM is knows as Store Description Language (SDL). The XML schema of CDL and SDL are later described in this section. To clarify, the landscape of models of the CIM framework is illustrated in Figure **??**. In addition, the well-formedness of the store model in CIM framework is also discussed in Section 3.5.

In CIM, a mapping set associates the Conceptual Visual Language to Store Visual Language which is later compiled into a set of Schema Description Language views. This compilation algorithm is presented in the next chapter. One of the characteristics of a multidimensional model is summarizability, which guarantees the accurate aggregate

CIM Visual Model (CVM)	**CIM Data Model (CDM)**
CVL : **C**onceptual **V**isual **L**ang.	**CDL** : **C**onceptual **D**ata **L**ang.
MVL : **M**apping **V**isual **L**ang.	**MDL** : **M**apping **D**ata **L**ang.
SVL : **S**tore **V**isual **L**ang.	**SDL** : **S**tore **D**ata **L**ang.

Figure 3.1: The landscape of models of the CIM Framework [67]

navigation during query answering. However, it turns out that the conceptual model in CIM might be non-summarizable after the mapping of conceptual and store models. In order to support summarizability in the CVL, we propose a technique, which is a data-driven refactoring mechanism. It analyzes the data and creates summarizable data models based on existing ones.

3.1 Motivating Example

Dream Home [15] as introduced in chapter 2, is an imaginary property management company where brokers mediate between potential buyers and owners who wish to buy or sell properties. The organization has branch offices in cities throughout North America. The company makes strategic decisions based on the analysis of a few numerical measures such as staff commission and revenue from selling a property. Figure 2.14 illustrates a small fragment of the Dream Home data warehouse expressed in the Conceptual Integration Model framework [67]. The Conceptual Visual Language is depicted on the left-hand side of the figure, while the Store Visual Language (a physical model of the data) is shown on the right-hand side. The SVL models integrity constraints among tables, which are represented by the intra-model dashed arrows. The CVL provides an abstract, high-level view of the data stored in the physical tables of the SVL.

In the CVL of Figure 2.14, *Branch Location* and *Client* (shadowed rectangles) are dimensions describing measures in the *Sale* fact relationship (shadowed diamond). Non-

shadowed rectangles (e.g., *Branch*, *City*, and *Client*) represent levels in the dimensions. These levels are organized into hierarchies by parent-child relationships, which are drawn as arrows from more specific levels to more general levels. The hierarchy in *Branch Location* indicates that company branches roll up to *City*, *Country* and *Region*. However, some cities roll up to *Province* (e.g., in Canada), some roll up to *State* (e.g., in the USA), and yet others roll up directly to *Country* (e.g., Washington DC).

Although CVL (conceptual) and the SVL (physical) models encapsulate the same data, they rely on different schemas. The representational gap between the CVL and the SVL models is filled by specifying *correspondences* between attributes of entities in the different models. For instance, the SVL shows that cities are physically stored in two different tables, *SCity* and *STown*. In Figure 2.14, the attribute mapping between store tables and the CVL construct *City* is given by the arrows linking the *cityID* and *name* attributes and columns in the three entities. These user-defined correspondences, which can also include value conditions, are part of the Mapping Visual Language (MVL). Mapping Visual Language has an XML representation, stored in Mapping Description Language (MDL). The CIM tool can compile the MVL into complex views over the physical model that can efficiently evaluate queries posed on the conceptual model. The process of compiling and rewriting these mappings as views over the data warehouse is addressed in the Chapter.

==========

3.2 Logical Representation of Conceptual Visual Language (CVL)

This section briefly describes the formal dimensional model of the conceptual model in CIM framework (see Figure 2.14 as an example). This model is the logical representation of the dimensions in the graphical CVL in Section 2.3.11. The logical representation of CVL model provides a formalism to flexibly model a data warehouse and infer different multidimensional properties of the data such as homogeneity and summarizability.

In CVL, data warehouses are expressed using a multidimensional model in which data are points (i.e., facts) in an n-dimensional space (i.e., data cubes). The *dimensions* of a data cube are the different perspectives needed to analyze data based on various *levels* of granularity. Levels are organized into lattices called *hierarchies*. A finer-granularity level *rolls up* into a coarser level to allow reporting at different levels of granularity.

A logical representation was formalized by Hurtado et. al [34] which expresses a multidimensional model for data warehouses. This framework captures both the schema and instance of dimensions. We adapt the model proposed by Hurtado et. al [34] as the logical representation of CVL in CIM framework. In this representation, a *level* is defined by a predicate symbol (the level name) and a set of arguments representing its attributes. Consider the finite sets $\mathcal{V}$ of domains, $\mathcal{A}$ of attributes and $\mathcal{L}$ of levels such that each entity $l \in \mathcal{L}$ of arity n is defined as $l = \{A_i : V_i \,|\, A_i \in A, V_i \in V, 1 \leq i \leq n\}$. A *dimension schema* D consists of a set of hierarchies. For each hierarchy $H \in D$, H is a tuple$(L, \nearrow)$, where L is a collection of levels (including a single top level *All* reachable from all levels in the hierarchy) and $\nearrow$ is a parent-child relation of two levels in L such that an element in the finer level rolls up to an element in the coarser level. One level could participate in different hierarchies and there could be more than one

top level, thus more than one hierarchy,in a dimension. We denote $\nearrow^*$ as the transitive and reflexive closure of $\nearrow$. Since level *All* can be reached by any level in the hierarchy, for each level $l_i \in L$ we have $l_i \nearrow^* All$. Note that each hierarchy has a single bottom level. Shortcuts are allowed in hierarchies, i.e., for a pair of adjacent levels l_i and l_j in a hierarchy ($l_i \nearrow l_j$), there could be an intermediate level l_k such that $l_i \nearrow^* l_k$ and $l_k \nearrow^* l_j$.

The instances of a hierarchy $H \in D$, $H = (L, \nearrow)$ are specified by a set of partial roll up functions: for each pair of levels (l_i, l_j), $l_i \nearrow l_j$, there exists a roll-up function $RUP_{l_i}^{l_j} : dom(l_i) \rightarrow dom(l_j)$. The sets $dom(l_i)$ and $dom(l_j)$ enumerate the elements of levels l_i and l_j, respectively, and an $RUP_{l_i}^{l_j}$ function returns the element in level l_j to which an element in level l_i rolls up.

Example 3.2.1 *The measures stored in the* Sale *fact table of Dream Home in Figure 2.14 are totalRevenue and staffCommission for a property sale contract with a client closed in a specific branch of the company. The* Branch *dimension schema shown in Figure 2.14 has* BranchLocation *hierarchy. It is described by* L = {Branch, City, Province, State, Region, Country }, $\nearrow$= {(Branch, City), (City, Region), (City, State) , (City, Province), (City, Country), (State, Country), (Province, Country), (Region, Country), (Country, All)}. *The bottom level for this hierarchy is* Branch. *The schema of the* Sale *fact relationship has two measures,* totalRevenue *and* staffComission.

All hierarchical data is located in the levels and parent-child relationships (roll up functions). There are no data instances for hierarchies and dimensions; instead, they organize the data in the levels and the roll up functions. The key role of the hierarchy and dimension elements is to specify how these components are structured in the multidimensional format.

3.3 Properties of the Conceptual Visual Model (CVL)

In this section we discuss some well known characteristics of dimension schemas in the multidimensional world. These properties are restated in the context of the logical representation described previously.

As mentioned in Section 3.2, parent-child relations between elements of two levels l_i and l_j are represented by a partial roll up function $\Gamma_{l_i}^{l_j} : dom(l_i) \rightarrow dom(l_j)$. This function associates elements of level l_i to their parent elements elements in level l_j. For a triple of levels, $l_i, l_j, l_k \in L$, in a hierarchy such that $l_i \nearrow l_j$ and $l_j \nearrow l_k$, we assume that $codom(\Gamma_{l_i}^{l_j}) \subseteq dom(\Gamma_{l_j}^{l_k})$. A path between two levels l_s and l_e is a sequence of levels $l_s, l_1, \ldots, l_k, l_e$ such that $l_s \nearrow l_1$, $l_i \nearrow l_j$ where $1 \leq i < j \leq k$ and $l_k \nearrow l_e$. Consider the defined path from l_s to l_e; according to the definition of roll-up functions, the element x in l_s rolls up to element $\Gamma_s^1 \circ \ldots \circ \Gamma_{k-1}^k \circ \Gamma_k^e(x)$ in l_e. Therefore, for each pair of paths in a hierarchy instance, $\tau_i = (l_1, l_2, \ldots, l_{n-1}, l_n)$ and $\tau_j = (l'_1, l'_2, \ldots, l'_{m-1}, l_m)$, where $l_n = l'_m$ and $l_1 = l'_1$, we have $\Gamma_{l_1}^{l_2} \circ \ldots \circ \Gamma_{l_{n-1}}^{l_n} = \Gamma_{l'_1}^{l'_2} \circ \ldots \circ \Gamma_{l'_{m-1}}^{l'_m}$. In fact, the properties of roll up functions and their compositions affect the property of dimension schemas and instances. The following definitions elaborate on this issue.

Definition 3.3.1 *A dimension instance is* **strict** *iff for every pair of levels l_s, l_e in each hierarchy schema, such that $l_s \nearrow^* l_e$ through more than two different paths, e.g. $\tau_i = (l_s, l_1, \ldots, l_n, l_e)$ and $\tau_j = (l_s, l'_1, \ldots, l'_m, l_e)$, we have that for every element x in l_s the following condition holds on the composition of roll up functions: $\Gamma_{l_s}^{l_1} \circ \Gamma_{l_1}^{l_2} \ldots \circ \Gamma_{l_n}^{e}(x) = \Gamma_{l_s}^{l'_1} \circ \Gamma_{l'_1}^{l'_2} \ldots \circ \Gamma_{l'_m}^{e}(x)$.*

In other words, a dimension instance is strict iff every element of the dimension has merely one ancestor element in each of the ancestor levels [9]. In fact, the strictness is defined as a constraint on the data which restricts the values that roll up functions can

take. Therefore, strictness of the dimension is a matter of consistency of data and it can not be considered as a property of the dimension schema. As a result, assuming that the underlying data is consistent and trustable, the strictness is satisfied automatically. Generally, in CIM, dimension instances are required to be strict, i.e., every element of a level should reach no more than one element in each ancestor category.

Definition 3.3.2 *A dimension is said to be* **homogeneous** *iff the domain of each roll-up function,* $\Gamma_{l_i}^{l_j} : dom(l_i) \rightarrow dom(l_j)$*, defined on level* l_i *is total. As a result, all possible compositions of roll up functions along the paths originating from a level* l_s*, e.g.* $\Gamma_{l_s}^{l_1} \circ \Gamma_{l_1}^{l_2} \cdots \circ \Gamma_{l_k}^{l_e}(x)$*, are total and covering all elements of the child levels.*

In a homogeneous dimension, all pairs of elements of a given level have ancestors in the same set of levels [34]. If all roll up functions are restricted to be total, then the compositions of them along all possible paths are also total. Thus, elements of the same level do not end up to elements of different parent levels. Forcing the schema to be homogeneous results in having big number of levels. By relaxing this restriction, we have dimensions that represent practical situations in a more natural and clean fashion. On the other hand, heterogeneous dimensions result in some difficulties in one of the basic OLAP mechanisms: computing an aggregate cube from other precomputed cube views. This issue is discussed in the following section, with more detail.

3.3.1 Summarizability

In order to introduce the notion of summarizability, we need to extend the formal dimensional model introduced previously with the specification of facts and data cubes.

A **fact relationship** F is a tuple $F = (\{D_1, \ldots, D_k\}, \{A_1 : V_1, \ldots, A_m : V_m\}, \{agg\})$ such that $A_i \in A$, $1 \leq i \leq m$, is a measure in the multidimensional model which is

described by dimension schemas $D_1, \ldots, D_k$ and aggregated by function agg. A **fact relationship instance** evaluating a fact relationship $F = (\{D_1, \ldots, D_k\}, \{m_k\}, \{agg\})$ is defined as a function $FR : (dom(l_1), \ldots, dom(l_k)) \rightarrow dom(m_k))$, where elements in bottom levels l_1, ... and l_k of dimensions D_1, ... and D_k, respectively, describe the values of measure m_k. A *data cube* is a fact relationship instance where the granularity of the dimension instances is not necessarily at the bottom level. In this case, the measure value is an aggregated value based on the aggregation function agg specified in the fact relationship..

Having a multidimensional model instance, $F_m^{l_a}(x)$ represents the aggregated value of measure m, in the fact relationship F, corresponding to element x in a single-level cube with granularity of level l_a. As originally stated, summarizability refers to whether a simple consolidation (aggregate) query, against a particular database instance, correctly computes a single-level cube view from another precomputed single-level cube view. This statement can be extended to defining the summarizability of a single level from a set of levels.

Definition 3.3.3 *In hierarchy* H*, a dimension level,* l_a*, is* **summarizable** *from level* l_b*, assuming* l_1 *as the bottom level of H, iff for every element* x *in level* l_a*, we have* $F_m^{l_a}(x) = \forall\ y_i \in dom(l_k)$ agg $F_m^{l_k}(y_i)$*, where* l_k *is any level along a possible path originating at the bottom level* l_1 *and ending at* l_a*,* $\tau_i = (l_1,\ldots,l_k,\ldots,l_a)$*. Also,* $x = RUP_{l_k}^{l_{k+1}} \circ \ldots \circ RUP_{l_{a-1}}^{l_a}(y_i)$ *along a possible path originating at level* l_k *and ending at* l_a*,* $\tau_i = (l_k,\ldots,l_{a-1},l_a)$*.*

The intuition behind this definition is that, in order for l_a to be summarizable from l_b, we need that the single-level cube specified for measure m with l_a granularity, be computable by summing up all cell values of single-level cube specified for measure m with l_b granularity. The concept of summarizability in a hierarchy can be derived from

a level summarizability.

Definition 3.3.4 *A hierarchy* H *is* **summarizable** *if for all levels* l_i, $1 \leq i \leq n$ *of this hierarchy, the single-level aggregated cube of measure* m *with* l_i *granularity, can be computed by summing up cell values of single-level cube specified for measure* m *for any* l_k *granularity,* $1 \leq k \leq n$, *appearing along a path from the bottom level to* l_i.

In other words, in a summarizable hierarchy the aggregated values for a measure at a level granularity can be obtained by aggregating the elements of any level of hierarchy which directly or indirectly rolls up to the desired level.

3.3.2 Representation of Mapping Visual Language (MVL)

In order to relate conceptual constructs to store objects, CIM uses attribute-to-attribute correspondences much similar to those in data integration [45] or data exchange [41]. Figure 2.14 shows the correspondences as lines drawn from conceptual level attributes (on the left) to the multidimensional level columns (on the right). In particular, the association of level attributes and table columns is of one to many multiplicity. Furthermore, conditions on the values of participating columns are allowed to be added to the correspondences.

Definition 3.3.5 *Given a CVL* C *and an SVL* S, *a* correspondence *between elements* c *in* C *and* s *in* S *is a tuple* $(C_i : a_j, \{S_1 : a_m, \ldots, S_k : a_n\}, \{cond_1, \ldots, cond_k\})$, *where the attribute* a_j *of XML element* C_i *in* C *is mapped to all attributes* a_l, $m \leq l \leq n$, *in XML elements* S_p, $m \leq p \leq n$ *in* S. *Each of these associatings is done under the condition* $cond_p$, $m \leq p \leq n$, *respectively. A constraint can be* null *or include conjunction and disjunction of equalities/inequalities of an attribute to constants or columns. The first element of the pair is called the* **head** *of correspondence* m, *the second element is the* **body** *of* c *and the third is called the condition of correspondence* c.

The fact that the second element of a correspondence pair is defined as a set denotes that an attribute in CVL can be mapped to a set of columns in SVL. In other words, the values of a level attribute can be provided by different columns. The correspondences can be interpreted as the association between values in SVL columns that have been specified to (virtually or physically) populate a CVL level attribute. This interpretation can be formally expressed by a source-to-target *tuple generating dependency* (tgd) [8]. A tgd for the correspondence association *cityID* attribute in level *City* to *cityID* columns in *SCity* and *STown* is as follows:

$$\forall\ \underline{c}\text{,n,d } \textbf{City}(\underline{c}\text{,n,d}) \rightarrow \exists\ \text{n}'\text{,cl } \textbf{SCity}(\underline{c},\ \text{n}',\ \text{cl}) \vee \exists\ \text{m}'\text{,cl}'\ \textbf{STown}(\underline{c},\ \text{m}',\ \text{cl}')$$

This is a correspondence equivalent to *City.cityID,* {*SCity.cityID, STown.cityID*}*,* {*null,null*} mapping which describes there must be a *SCity* ot *STown* tuple for each *City* element, which carries the same *cityID* as the city's *cityID*. The variable n represents the correspondence between the CVL attribute and SVL columns.

The correspondences do not contain any information about the semantic relation of data values inside the CVL or SVL objects. For instance, correspondences connecting *cityID* and *name* elements in *City* to columns of *SCity STown* do not carry information about how a *City* tuple is actually build. However, it is not practical for a high-level user accustomed to only the conceptual view of the data to deal with the complexity of integrating correspondences into a more meaningful mapping. The solution of CIM requires the user to only provide *correspondences* between attributes and columns of CVL and SVL, which are later transparently compiled by the system into complex, fully-fledged mappings that can be used as views over the underlying data warehouse.

In fact, CIM uses mappings of the form $c \rightsquigarrow \psi_s$, where c is an element in the *target* schema and ψ_s is a query (or view) over the *source* schema. In CIM framework, the conceptual model (CVL) functions as the target schema and the store model (SVL) as

the source schema. The element c in the mapping expression is either a level, a parent-child relation, or a fact relationship, whereas ψ_s is a view over the data warehouse tables. CIM utilizes *sound* mappings, which are defined by $\forall\bar{x}(\psi_s(\bar{x}) \rightarrow c(\bar{x}))$ expressions, where $\bar{x}$ denotes a set of variables.

These mappings are represented as views built upon the data warehouse modeled in SVL. Thus, they can be generated with any standard view definition language such as SQL.

3.4 XML Representation of CDL, SDL and MDL

Users can simply model the concepts of a multidimensional environment and their mappings to the data warehouse, using the CIM Visual Model. To facilitate integrating with BI applications, visual components — CVL, MVL, and SVL — are translated into their equivalent XML-based object model — CDL, MDL, and SDL, respectively. In this XML representation, each construct of CVL, MVL, and SVL is described by an XML Schema type. Therefore, a user can design concepts in CIM graphical environment that are automatically translated into XML models for run-time usage.

For each main construct in CVL, SVL and MVL, there is one type in their corresponding XML models. The root of the XML model for CDL is a complex type that pertains to property, level, dimension, level, factRelationship, and sometimes hierarchy. Each of these sets contains a list of all constructs that appear in a given model (levels, dimensions, fact relationships and hierarchies, respectively). Figure 3.2(a) shows the CDL of some constructs from the visual models in Figure 2.14. Level City is shown as a type which contains property types specifying its attributes. The attributes type and nullable provide more information about level properties. Moreover, the key attribute of City level is separately defined as a key type.

As mentioned previously, the SDL model shows how the warehouse is physically stored. The SDL model is built on a relational approach and has two XML schema types: **table** and **column**. By convention, each relational table should have a primary key and may have a set of foreign keys referring to other tables. The SDL model root is a complex type containing sets of fact tables and dimensions. Primary and foreign keys of a table are reflected as **key** and **foreignkey** types within a **table** type. Figure 3.2(b) shows the SDL of some SBranch and SCity tables (STown has the same schema as SCity), in SVL of Figure 2.14. The SBranch dimension table possesses eight columns, as shown in Figure 2.14. In addition to the name, each column has self-explanatory type and nullable attributes. One of those columns is a primary key (denoted by the key element) and four are foreign keys, identified by the foreign key elements.

Each correspondence provided is represented, in MDL, as one or more *property mappings* between the CDL and SDL models. A property mapping is a one-to-one attribute association and a correspondence, as previously described, can span several of them. For instance, in Figure 2.14 there is a single correspondence for cityID between the CVL City and the SVL SCity and STown. Such correspondences are composed of two different property mappings for cityID: one between City and SCity, and the other one between City and STown. This type of correspondence that spans more than one store entity is called a *disjunctive correspondence*.

The MDL schema defines mappings between CDL and SDL elements. The attribute-to-attribute correspondences provided by the users in the MVL are grouped into *mapping fragments*. A mapping fragment consists of all attribute associations between one CDL entity and one SDL table. A mapping fragment is represented in MDL as a level-mapping or factrel-mapping element containing the names of the CVL entity and the SVL table being connected. Each mapping fragment contains a set of one or more property-mapping

```
- <level name="City">
    <key name="cityID" />
    <property name="cityID" type="String" nullable="false" />
    <property name="name" type="String" nullable="true" />
    <property name="description" type="String" nullable="true" />
  </level>
- <level name="State">
    <key name="stateID" />
    <property name="stateID" type="String" nullable="false" />
  </level>
```

(a) CDL of City and State entities

```
- <dimension name="SBranch">
    <key name="branchID" />
    <foreignkey name="city1" table="SCity" column="cityID" />
    <foreignkey name="city2" table="STown" column="cityID" />
    <foreignkey name="state-province" table="SStateProv" column="spID" />
    <foreignkey name="region" table="SRegion" column="regionID" />
    <column name="branchID" type="nvarchar" nullable="false" />
    ...
  </dimension>
- <dimension name="SCity">
    <key name="cityID" />
    <column name="cityID" type="nvarchar" nullable="false" />
    <column name="name" type="nvarchar" nullable="false" />
    <column name="population" type="nvarchar" nullable="true" />
    <column name="countryName" type="nvarchar" nullable="true" />
  </dimension>
```

(b) SDL of SBranch and SCity tables

```
- <level-mapping level="State" dimension="SBranch">
    <property-mapping property="stateID" column="state-province" />
    <condition column="isProvince" expression="false" />
  </level-mapping>
- <level-mapping level="City" dimension="SCity">
    <property-mapping property="cityID" column="cityID" />
    <property-mapping property="name" column="name" />
  </level-mapping>
- <level-mapping level="City" dimension="STown">
    <property-mapping property="cityID" column="cityID" />
    <property-mapping property="name" column="name" />
  </level-mapping>
```

(c) MDL of State and City mapping fragments

Figure 3.2: Dream Home CIM Data Model Fragment

elements and optional condition elements. Each property-mapping element represents one of the user-defined associations between a CVL property and an SVL table column. The condition elements specify the attribute values for which the mapping fragment holds. Figure 3.2(c) demonstrates the correspondences defined for attributes of *City* and *State* levels.

3.5 Well-formed Data Warehouse

The Store Description Language does not impose any restriction on how dimension and fact tables are connected in a data warehouse. However, data warehouses are not arbitrary relational models, consisting of tables and referential integrity constraints. In practice, fact and dimension tables follow certain patterns [46, 54]. Such schema patterns vary from one fact table and one table per dimension (star schema) to multiple normalized dimension tables (snowflake schema) and multiple fact tables sharing dimension tables (constellation). Hybrids in which data has been partially normalized (starflake schemas) or organized in standard tables (data-vault) are also common. These schemas were briefly introduced in Chapter 2. In spite of various schemas used for data warehouses, they can be classified according to the characteristics of multidimensional models. We formalize next the notion of a *well-formed* SDL schema that captures all the patterns mentioned above:

Definition 3.5.1 *Let $G = \{N, E\}$ be the complete schema graph of an SDL schema S. We distinguish two sets of nodes in N, $D = \{n \in N \mid \exists s \in S : \chi(s, n)$ and s is a dimension table$\}$ and $F = \{n \in N \mid \exists s \in S : \chi(s, n)$ and s is a fact table$\}$. We also distinguish a subset of edges in E, $E_D = \{\langle d, d' \rangle \in E \mid d, d' \in D\}$. We call the subgraph $G_D = \{D, E_D\}$ a* dimension graph. *Each edge $e = \langle d, d' \rangle$ in E_D is associated with a roll-up function $\Gamma_d^{d'}$ such that $\Gamma_d^{d'}(x) = y$ iff $\chi(t, d)$, $\chi(t', d')$, $x \in t$, $y \in t'$, c is a referential integrity constraint from t to t' s.t. $\chi(c, e)$ and (x, y) satisfies c. We say that SDL schema S is* **well-formed** *if it satisfies the following:* (1) *G is a DAG;* (2) *all roll up functions $\Gamma_d^{d'}$ associated with edges in E_D are total, i.e., every instance of level d is assigned an instance of level d' by $\Gamma_d^{d'}$;* (3) *G_D is partitioned into* dimension lattices *that are pairwise disjoint (i.e., they have no node or edge in common) and whose union is G_D;* (4) *each dimension lattice has a unique top node and a unique bottom node;* (5)

for every edge $\langle n, d\rangle \in E$, $d \in D$; *and* (6) *for every* $\langle f, d\rangle \in E$, $f \in F$, $d \in D$, d *is the bottom of some dimension lattice.*

Conditions (1) and (2) are necessary for supporting summarizability. A dimension is *summarizable* if aggregations at finer levels in a hierarchy can be used to compute aggregations at higher levels, thus increasing the efficiency of answering roll up queries. As discussed in [64, 56], a necessary condition for summarizability is to have *total* roll up functions, which is similar to the notion of dimension homogeneity in [33]. A non-summarizable dimension can be converted to one or more summarizable ones by either fixing their instances [56] or by refactoring their schemas [60]. For instance, the example in Figure 2.14 is not summarizable: the roll up functions from City are not total (some cities roll up to Province and some others to State). This hierarchy can be easily made summarizable by splitting it into two independent and summarizable hierarchies: Branch-City-*State*-Country and Branch-City-*Province*-Country. Once this is done, aggregations pre-computed at finer levels can be used to compute aggregations at coarser levels of the hierarchy. Conditions (3) and (4) state that dimensions (comprised by one or more dimension tables) are disjoint and that they have a unique top and bottom. Finally, condition (5) forbids edges between two fact table nodes and condition (6) specifies that every fact table is connected to the bottom of some dimension lattice.

3.6 Schema Description Language Views

The attribute-to-attribute correspondences defined by users between conceptual and store constructs do not express any information about the structure of constructs and tables association within a schema. The CIM conceptual and store models are related by mappings similar to those in data integration [45] or data exchange [41]. In particular,

CIM uses mappings of the common form $c \rightsquigarrow \psi_s$, where c is an element in the *target* schema and ψ_s is a query (or view) over the *source* schema. In CIM, the conceptual model (CDL) functions as the target schema and the store model (SDL) as the source schema. The element c in the mapping expression is either a level, a parent-child relationship, or a fact relationship, whereas ψ_s is a view over the data warehouse's multidimensional tables. CIM uses *sound* mappings, i.e., those defined by $\forall \bar{x}(\psi_s(\bar{x}) \rightarrow c(\bar{x}))$ expressions, where $\bar{x}$ denotes a set of variables.

Since it is not practical for a high-level user accustomed to only the conceptual view of the data to come up with complex view definitions in terms of the tables of the physical data warehouse. Our solution requests the user to only provide very simple property mappings between attributes in both models; these are later transparently compiled by the system into complex, fully-fledged, multidimensional mappings that can be used for query evaluation.

3.7 How to Build Summarizable Dimensions

The notion of summarizability was first introduced by Rafanelli and Shoshani [66] by identifying two necessary conditions for summarizability. The first condition which should be tested at the instance level, is the disjointedness of level attributes. This condition states that, in a parent-child relationship, the level attributes of the more general level must cluster the elements set of the more specific level into disjoint levels [43]. Given that this condition is satisfied, the second condition checks if this clustering is complete: (1) the union of all elements in the clusters constitute the whole set of the specific level, and (2) each specific element is assigned to a general element. In brief, these two conditions constrain the parent-child relationships to be partitioning functions on finer levels. In [43], the same conditions along with a new important condition were applied

to elements of the multidimensional space, to provide a rich framework for determining summarizability. The third summarizability condition depends on the type of measure attributes and aggregate functions. For instance, there are functions that are inherently non-summarizable, such as ‘’median‘‘ and ”’percentiles”’ or ”‘trimmed mean”’. In such cases one has to generate the summary values directly from the data in the bottom level of the hierarchy. In [43], it was informally argued that these three conditions together are sufficient for summarizability.

The idea of OLAP aggregate navigation had been presented in the early 1980's in graph models for statistical data bases (e.g., in ([66]), when the the notion of cube view dependence first introduced by Harinarayan et al. [28]. They express the dependencies among views in the form of a lattice framework and present greedy algorithms that work off this lattice and determine a good set of views to materialize.

However, in many real-world applications, the data fail to comply with these rigid constraints. Traditional OLAP data managers do not model the non-summarizability prompted by the heterogeneity of data; they treat the heterogeneity by forcing designers to add null values, which may cause an exponential growth of the aggregates handled in data cubes, adversely affecting storage space and time spent on maintenance. This motivated the research for techniques that support summarizability. Two approaches have been investigated in this challenge: (1) multidimensional modeling methodologies that address how to conceptually model complex multidimensional structures such that query responses are possible by enforcing summarizability. (2) algorithms to automatically achieve summarizability for underlying heterogeneous data. As an example of the first approach, in [44], a set of constructs (level-level association, level-level generalization and fact-dimension association) has been listed for multidimensional modeling, together with a set of rules, on the multiplicity of associations, that imply summarizabil-

ity. The second approach discusses two issues in multidimensional systems [68]: allowing the modeler to define complex structures on hierarchies, in a explicit way, and providing mechanisms to the data management system to avoid summarizability problems. Some other methods (e.g., [63], [30] and [9]) transform irregular dimension hierarchies and fact-dimension relationships into well-behaved structures that when used by existing OLAP systems, enable summarizability and, thus, practical pre-aggregation. These methods revolve around two key ideas: (1) finding repairs and modifications at the multidimensional data level without changing the schema of data, (2) rewriting the data schema in a summarizable form, (3) consistent query answering. Following the first idea, Pedersen et al. [63] propose transforming a particular class of heterogeneous dimensions into balanced dimensions by adding null elements to represent missing coarser values. On the other hand, in the work done by Hurtadi et al. [34], introduced a new class of integrity constraints on stored data as dimensions in order to overcome the limitation that the hierarchy schema itself is not expressive enough to support summarizability reasoning in the presence of heterogeneity. These schemas extend classes of OLAP dimension schemas, such as the model of Jagadish el al. [35] which partially solves the limitations of traditional OLAP models by allowing several bottom categories. Hurtado et al. [31][30], in subsequent research, introduced the notion of frozen dimensions which are minimal homogeneous dimension instances representing the different structures, combined implicitly in a heterogeneous dimension. The notion of frozen dimensions provides the basis for efficiently testing the implication of dimension constraints. In addition, they are essential for studying algorithmic aspects of summarizability and generating pre-aggregated views in dimension schemas. In a different approach, Bertossi et al. [9], defined the notion of consistent answer to an aggregate query with group-by statements. Also, they present the canonical dimension that allows the system to compute approximate answers.

While most multidimensional modeling approaches exclusively focus on satisfying summarizability of dimension hierarchies, a few works (e.g., [40] and [55]) address summarizability issues in fact-dimension relationships. Multi-valued dimensions [40] allow a star schema to have non-strict relationships between facts and dimensions by using a bridge table. The primary key for this bridge table consists of the foreign keys referring to the dimension and fact tables. The approach proposed by Mazon et al., [55], provides an approach at the conceptual level for identifying problematic fact-dimension relationships and applies a normalization process to transform the model into a summarizable model that avoids erroneous analysis of data.

As discussed, summarizability is considered as a necessity in any application that supports OLAP operations. However, dimension schemas and instances that contradict this property are contingent. Therefore, in order to have a realistic practical aggregate navigation as well as accurate query answering, the dimension schema should allow query answering process to access summarizable data units.

As mentioned previously, the homogeneity property enforces elements of a given dimension level to roll up to its ancestors in the same set of levels. Assuming that few elements of level l_i roll up to different parent levels, in order to compute the aggregated cube for one of the parent levels, all elements of level l_i are considered as well as those of other child levels. This results in an incorrect aggregated value since the elements of l_i that do not roll up to the considered parent participate in the aggregate computation process. Intuitively, heterogeneity of data elements in a dimension schema can cause non-summarizability in computing aggregated cubes. According to Pedersen et. al [63], summarizability occurs when: (1) the roll-up functions in the hierarchies are covering (all child elements can be related to elements in the same parent), (2) the roll-up functions are strict (each child element in a hierarchy has only one parent element in a particular

level), (3) the relationships between facts and dimensions are many-to-one and facts are always mapped to the lowest levels in the dimensions. In Section 3.5, it was explained that the strictness criterion constrained the dimensions not to have inconsistent data, that is, an element rolls up to the same element in its ancestor taking different paths. However, this rule is tied to the consistency of the data warehouse. In CIM, we presume that the underlying data satisfies this property and, thus, condition (2) of summarizability. The last condition mentioned by Pedersen et. al is assumed to be satisfied in the conceptual model of the CIM framework, since a fact relationship is always associated to the bottom level of different dimensions. Therefore, in the CIM framework, the necessary condition for a conceptual schema instance to be summarizable is that all roll-up functions in the model be total (covering) functions. This argument is formalized in the definition of summarizability3.3.3. However, as it is allowed to define more than one parent-child relation on a level, apparently some roll-up functions are not total, or in other words turn out to be heterogeneous.

Definition 3.7.1 *Given a dimension model D and its instance, hierarchy H,* $H = (L, \nearrow)$*, with bottom level* l_b*, is* summarizable *iff all roll-up functions* $RUP_{l_i}^{l_j}$*,* $l_i, l_j \in L$ *are total.*

According to this definition, the summarizability issue is to tighten up to the homogeneity of data. To illustrate, the following example describes the problem that is observed in computing aggregated data for a heterogeneous dimension model.

Example 3.7.1 *Assume that a query on the conceptual model in Figure 2.14, requests for the total revenue of lease contracts handled in branches at the country* USA*; the total revenue has already been calculated and stored for* State, Province *and* City *levels. If the requested total revenue is computed by merely considering the parent-child relation between the levels* State *and* Country*, the resulting value covers the cities to which a state*

has been assigned. This results in having the revenue in all cities except Washington*, which does not belong to any state. This issue originates from the heterogeneity of data, where different cities in* City *level have parents in various levels (*State *and* Country*).*

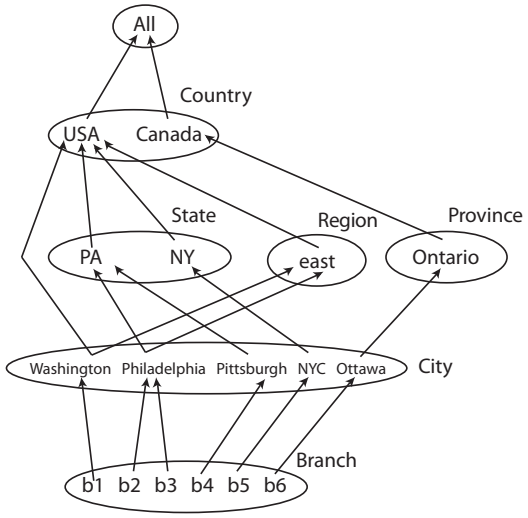

Figure 3.3: A non-summarizable dimension schema and instance.

In fact, the source of the non-summarizability issue discussed in Example 3.7.1 is related to the way data is structured in levels. As explained in Definition 3.7.1, non-summarizability occurs in a hierarchy when there are elements in a level which have different corresponding elements in parent levels. For instance, in Figure 3.3, which is an instance of *BranchLocation* hierarchy in Figure 2.14, the city $Washington$ rolls up to an element in level *Country*, while *Ottawa* and the rest are rolling up to *Province* and *State* elements, respectively.

Refactoring of a hierarchy such that the data in the schema satisfies the summarizability condition leads to correct aggregation in such queries. One way to achieve such a hierarchy model is to split the schema and data of a hierarchy such that all elements of each level roll up to elements of all parent levels. The hierarchy in Figure 3.3 is not summarizable because branches belong to either a state or a province (or in case of

Washington exceptionally directly to the country) according to the country of location. Intuitively, the solution of this specific issue is to refactor the schema and data elements in *Branch* level into three different classes: (1) the branches which roll up to *State*, (2) the branches which roll up to *Province*, and (3) the branches which roll up directly to *Country*. Each of the classes of *Branch* level can be the bottom level of a new hierarchy. The schema of these hierarchies are also modified according to the data in the bottom level. For example, a level holding the branches of *Washington* does not have *State* and *Province* levels as its parent level, in addition, the countries other than *USA* are not included in the *Country* level of such hierarchy. In other words, the separation of elements in any level is propagated to the elements of its ancestors, thus, possibly changing the schema of the hierarchy. In addition to the schema adaptation, the elements of intermediate levels are modified according to their children and parent elements, in order to have all level elements rolling up to the same levels. Note that the data elements in the bottom level are partitioned, whilst it is possible to have repetitive elements in other levels, due to the fact that more than one element can have the same parent element.

Definition 3.7.2 *Given a dimension model D and its instance, a subhierarchy of hierarchy H, $H = (L, \nearrow)$, with bottom level l_b, is $H' = (L', \nearrow')$, such that (1) $L' \subseteq L$, $\nearrow' \subseteq \nearrow$, (2) $l_b, All \in L'$, (3) the elements of level $l_i \in L'$ is a subset of its original elements in the main hierarchy, (4) all elements in l_b can roll up to level All taking all possible paths in the hierarchy graph.*

Working with summarizabile subhierarchies in a dimension model implies that the aggregated data of an ancestor level in a subhierarchy can be computed from the aggregated data of any child level of this particular level. As a result, if the subhierarchies of a dimension partition the data elements of the hierarchy bottom level, the aggregated values of a particular level is the union of aggregated values computed for each of its

subhierarchies. The partitioning idea prevents the duplication of data in the bottom level and therefore in aggregation.

3.7.1 Schema-Driven Methods

Partitioning a hierarchy to a set of summarizable subhierarchy models requires sufficient information about the organization of data in levels. Hurtado et. al [30] proposed a framework that models structural irregularities of dimensions by means of integrity constraints. This class of integrity constraints, dimension constraints, allows us to reason about summarizability in heterogeneous dimensions. They introduce the notion of **frozen dimensions** which are minimal summarizable dimension instances representing the different structures that are implicitly combined in a heterogeneous dimension. Dimension constraints provide the basis for efficiently generating summarizable hierarchies. The reason that restricts using this algorithm is the necessity of specifying integrity constraints, by a domain expert, as an aid to understanding heterogeneous dimensions. Following this traditional approach to dimension modeling, that is to expect dimension constraints, is not feasible in the CIM framework, as the modeling is done by high level users that do not deal with data. However, the knowledge about the organization of data is hidden in the elements of hierarchy levels. We propose an algorithm for refactoring a hierarchy model into a set of summarizable subhierarchies by solely processing the data elements of the model. In this algorithm, the Bisimulation [39] notion is utilized as a method of partitioning data in each level into classes with total roll-up relation with its parent levels. This approach scans the dimension data and generates a new set of hierarchies satisfying the summarizability criterion.

3.7.2 Data-Driven Methods

Pedersen et al. [63] provided transformation techniques that render dimension hierarchies that do not satisfy the summarizability property. In this approach, the transformation of the dimension hierarchies is performed in three steps. First, all roll-up functions are transformed to be total, by introducing extra intermediate elements. Second, all roll-up functions are transformed to be onto, by introducing placeholder elements at finer levels for values without any children. Third, roll-up functions are made strict by merging elements together. In general, this algorithm solves the non-summarizability problem by changing the data elements in levels' domain, as opposed to the method proposed in [30] which refactors the hierarchy schema.

3.7.3 Our Method: Dimension Refactoring

Two different approaches for summarizability satisfaction have been discussed. One revolves around refactoring the schema according to the comprehensive knowledge provided as constraints in a dimension schema. Whilst the other one repairs the data elements of a dimension instance in order to create a summarizable dimension instance. Here, we propose an approach which refactors a hierarchy schema, thus level elements, into a set of subhierarchies that satisfy the summarizability property. This refactoring is done by processing the hierarchy data instance and splitting the schema based on data relations.

3.7.4 Bisimulation

In this section, we briefly explain the notion of bisimulation, that can serve as a means of generating summarizable subhierarchies. In the context of set theory, the bisimulation problem is equivalent to determining the coarsest partition of a set, with respect to a given relation. The notion of Bisimilarity [39] is defined on graph structure of databases.

A hierarchy schema $H =< L, \nearrow >$ can be modeled as a directed graph hierarchy $D =< L, PC >$, vertices in L correspond to the levels of the hierarchy and every parent-child relation in $\nearrow$ is represented by a directed edge connecting two involved levels. We can extend this graph with the data elements in the hierarchy instance according to the mappings generated for levels during the mapping compilation process. This extension is represented as a directed labeled graph $G =< V_G, E_G, all, \Sigma_G, label, value >$. In this signature, Σ_G contains the name of hierarchy levels in L. Each level element in the hierarchy instance H is identified by a unique *id* by the *label* function. Each identifier, or level element, is a vertex in V_G which is labeled with the name of the level it belongs to, specified by *value* function from Σ_G. Each parent-child relation instance, that is expressed by the corresponding roll-up function, is an edge in E_G from the parent element vertex to the child vertex. Moreover, there is also a single element called *all* that is distinguished by label *ROOT*.

Example 3.7.2 *Figure 3.4 depicts the instance graph of BranchLocation hierarchy in Figure 3.3. In the graph, one node is defined for each element of the instance; the nodes are labeled by the value of corresponding element. They can also be indexed by numbers. The parent-child relation between two elements is reflected by an edge drawn from the parent element to the child one.*

Definition 3.7.3 *The notion of* **bisimulation** *is defined for graph* $G =< V_G, E_G, all, \Sigma_G, label, value >$ *as a binary symmetric relation on* V_G *[39]. Two vertices* u *and* v *in* V_G *have bisimulation relation, denoted by* $u \approx^b v$ *if (1) they have the same label assigned by the function value, (2) if their parents have a bisimulation relation.*

Based on this definition of bisimilarity and the directed labeled graph of hierarchies, in the corresponding graph G of a hierarchy model, two elements are bisimilar if they

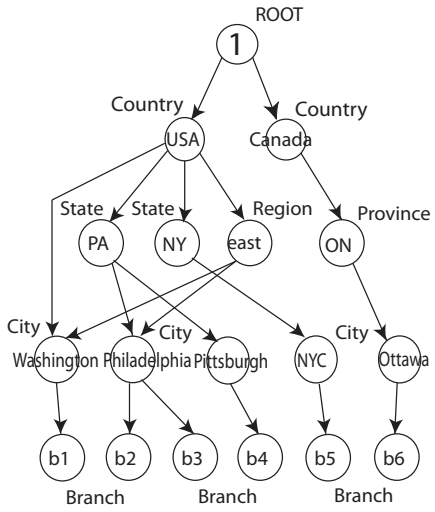

Figure 3.4: The data graph structure of *BranchLocation* hierarchy in Figure 3.3.

belong to the same level and their parent elements (if available) are also in the domain of one single level. For instance, *Philadelphia* and *NYC* nodes in Figure 3.4 are bisimilar since they belong to the same levels and *NY* and *PA* nodes are also bisimilar. The reason for two nodes *NY* and *PA* being bisimilar is that they are the elements of level *Province* and share the same parent which are bisimilar by convention. The association of bisimilar elements and their anecestors can be generalized as a schema that these elements represent one of its instances.

Definition 3.7.4 *An* **index graph** *can be created considering the partition of V_G inferred by $\approx^b$ relation. Each vertex of this index graph corresponds to a partition class and those vertices of graph G that are members of this class are considered as the extent or elements of this vertex. If there is an edge in G between element e_i of vertex U in the index graph to element e_j of vertex V, then an edge should be drawn from V to U in the index graph. The index graph of hierarchy instance graph G is called* **Bisim(G)** *or bisimilarity graph of G.*

Note that the edge is in the opposite direction just to indicate the roll-up relation.

The corresponding vertices of bottom level elements in *Bisim(G)* that do not have any in-going edge are called 'simple' vertices.

Definition 3.7.5 *Assume that the graph-structured model* $G =< V_G, E_G, all, \Sigma_G, label, value >$ *has been created for the hierarchy* $H =< L, \nearrow >$ *and its instance. The subgraphs of graph Bisim(G), made by traversing the graph starting from a 'simple' vertex to get to the Root vertex is called a* bisimilar subhierarchy *of H.*

In the specified subgraphs of graph *Bisim(G)*, each class is a new level having the vertex extent as its domain. In a bisimilar subhierarchy, the roll up function between two adjacent levels/vertices U and V is the same as the roll-up function defined in H. However, the domain and codomain of this roll-up function are restricted to corresponding elements of vertices U and V, respectively. Figure 3.5 depicts all bisimilar hierarchies of Figure 3.4 which represents the data graph of *BranchLocation* hierarchy. As it can be observed, all elements of each level roll up to the same levels; thus, the corresponding roll-up function is total on the domain of each level.

Proposition 3.7.1 *The bisimilar subhierarchies of a hierarchy model are summarizable.*

Proof 3.7.1 *Based on the definition of bisimilarity, two elements, in a data graph, are bisimilar if they have the same label and their parents are bisimilar as well. In a data graph representing an instance of a hierarchy, the vertices are labeled by the name of the level to which they belong. Therefore, bisimilar vertices are the elements of the same level. In addition, since their ancestors are also bisimilar, they belong to the same level. On the other hand, bisimilarity relation is a transitive relation; if* $u \approx^b v$ *and* $v \approx^b w$*, then* $u \approx^b w$*. As a result, all vertices that are bisimilar to a specific vertex are of the same level and have parents and ancestors of the same level. In the index graph of Bisim(G) all bisimilar elements are organized in a class (or level) and their*

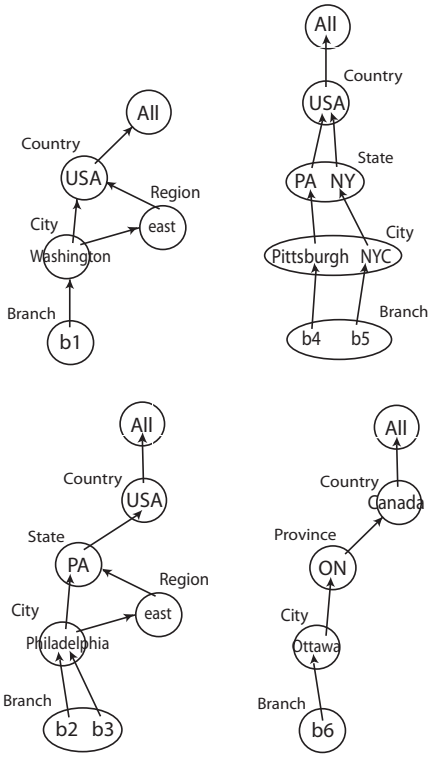

Figure 3.5: The set of bisimilar subhierarchies for data graph structure of *BranchLocation* hierarchy in Figure 3.4.

parent-child relation is inferred according to the edges of vertices. By a straightforward inference, it can be concluded that all elements of a level in the index graph have an adjacent element in the parent level. Consequently, when bisimilar subhierarchies are created by traversing the simple vertices of the index graph (see Definition 3.7.5) the roll-up functions of the corresponding subhierarchy are all total, thus the subhierarchy is summarizable (see Definition 3.3.3).

In summary, the instance of a dimension hierarchy can be expressed as a data graph which represents elements and their parent-child relations as labeled vertices and edges, respectively. As a second step, finding bisimilar subhierarchies of this graph will result in a set of summarizable subhierarchy models. Such subhierarchies are generated by scanning the data elements and applying a bisimulation algorithm in the literature. The bisimulation reduction of a labelled graph can be performed in polynomial time with respect to the size of the data and parent-child relations [62], or even linearly for acyclic data instance graphs, as shown in [20]. The resulting subhierarchies are the summarizable subhierarchies that are appropriate for query answering and aggregate navigation.

3.7.5 Query Answering using Summarizable Subhierarchies

Summarizability facilitates aggregate navigation as well as accurate query answering. In Section 3.7.3 we presented an algorithm which extracts summarizable subhierarchies for a hierarchy model. This method is data-driven to process the data warehouse and refactor the structure and instances of levels such that all level instances roll up to the same levels. A new schema is generated for each summarizable subhierarchy together with the data nodes of corresponding bisimilar graph as levels instances. The next step is to investigate the incorporation of the summarizable hierarchies into an aggregate navigation process. According to the definition of summarizability in Definition 3.3.3,

the aggregation value of a measure with respect to an element of a level is computed by performing the aggregation on all its children elements in different levels. Therefore, if the aggregated value of such elements has been computed in answering previous queries, the new aggregation can be done without extra computation. In a more formalized way, assume query Q is posed against hierarchy H with the set of summarizable subhierarchies $\{h_1, \ldots, h_n\}$. It requests the aggregated value of measure m with respect to value v_i of level l_a. The answer to this query is $F_{m,H}^{l_a=v_i} = agg_{1\leq i<n}\ F_{m,h_i}^{l_a=v_i}$, where $F_{m,H}^{l_a=v_i}$ is the total aggregated value of measure m for value v_i at level l_a of hierarchy H which is calculated by applying *agg* function over partial values of subhierarchies $\{h_1, \ldots, h_n\}$. Similarly, $F_{m,h_i}^{l_a=v_i}$ is the aggregated value of measure m for value v_i at level l_a. According to Definition 3.3.3, it is equivalent to $aggF_m^{l_k}(y_i)$, where l_k is any level of hierarchy h_i, $1 \leq i < n$ along a possible path originating at the bottom level l_1 and ending at l_a, $\tau_i = (l_1,\ldots,l_k,\ldots,l_a)$. Also, $x = RUP_{l_k}^{l_{k+1}} \circ \ldots \circ RUP_{l_a}^{l_a-1}(y_i)$ along a possible path originating at level l_k and ending at l_a,$\tau_i = (l_k,\ldots,l_{a-1},l_a)$.

Example 3.7.3 *Assume that a query on the conceptual model in Figure 2.14, requests for the total revenue of lease contracts handled in branches at the country* USA*; the total revenue has already been calculated and stored for* 'Philadelphia'*. A set of summarizable subhierarchies are generated and shown in Figure 3.5. The query is posed against all these subhierarchies and since the total value was requested, the returned values are summed up to obtain the accurate answer of the query. In hierarchy* h_1*, the aggregated value for* USA *is the measure value for* Branch='b1'*. A similar justification leads us to return the sum revenue of branches* b4 *and* b5*. In case of hierarchy* h_2*, since the total revenue at city* 'Philadelphia' *is already computed, the returned value is the existing value for this city. Finally, no total revenue is returned for hierarchy* h_4 *since this hierarchy is related to Canadian branches of Dreamhome. The sum of all partial revenues returned by different*

subhierarchies will be the answer to the query about the total revenue of Dreamhome in American branches. In a more complex case, where, for instance, the aggregated value of region east *has already been computed, since this region appears in two hierarchies* h_1 *and* h_2*, it covers the partial revenue in branches in* east *region. Therefore, it is not required to compute revenue in hierarchies* h_1 *and* h_2*.*

Intuitively, the calculation of an aggregated value for a measure with respect to a hierarchy, is achieved by aggregating the measure for relevant elements of the bottom level along all subhierarchies. However, in case of storing the aggregation values, they can be reused and the corresponding elements of the bottom level are not considered. The following proposition discusses the correctness of the answers to queries using the aforementioned approach for computing aggregated values.

Proposition 3.7.2 *The aggregation of partial aggregate values in summarizable subhierarchies of a hierarchy is the accurate aggregate value of that hierarchy.*

Proof 3.7.2 *In the data graph of a hierarchy, all level elements are represented as a vertex which is connected to its parent vertices. Applying Bisimulation algorithm on this graph categorize bisimilar vertices in the same class. According to the definition of bisimulation in Definition 3.7.3, it can be proved that the vertices corresponding to bottom level elements are partitioned into bisimilar subhierarchies bottom levels. Bisimilar subhierarchies are created based on index graphs, which, starting from simple vertices, categorize bisimilar vertices in the same level, and draw parent-child relation according to the edges of the graph. Applying Bisimulation, partitions the simple vertices set, meaning it assigns the vertices labeled with the same level, that have the parent vertices labeled with similar level name, to one class. No vertex is repeated in different classes, because if the members of the first class are bisimilar to this vertex, then the members of the second class are also bisimilar to the members of the first class. Thus, they should appear in*

the same class. This implies that no bottom level element of the hierarchy is missed or duplicated in the summarizable subhierarchies. As a result, if the aggregation is done on all elements of the bottom levels that roll up to a coarser element through a path, the same aggregate value is obtained.

Considering this proposition, we can claim that no element is duplicated in bottom levels of refactored subhierarchies, and the basics of our query answering algorithm is to aggregate all relevant bottom level elements of subhierarchies. Therefore, the answers to user queries utilizing this approach is correct.

Chapter 4

Mapping Compilation

Mapping compiler is a component in Conceptual Integration Model, which takes as input a pair of CDL and SDL schemas and an MDL mapping between them. Based on the mapping fragments and the integrity constraints in the schemas, the compilation process generates a set of views that map CDL schema elements to SDL schema elements. This ultimately allows users to query the conceptual model and have the queries rewritten in terms of views over the data warehouse where the actual data is stored.

As previously discussed, a property mapping associates one attribute in CDL to a column in SDL. Based on property mappings we can build MDL mapping fragments that are simple mappings relating a set of attributes belonging to one CDL entity to one SDL table. Different attributes of a CDL entity can be mapped to more than one tables, thus appearing in multiple mapping fragments. A correspondence is defined for each attribute in CDL and groups the property mappings that are specified for the attribute. If a correspondence contains more than one property mappings, it can be called as disjunctive correspondence. The compiled SDL views (Section 4.1) integrate conceptual elements that are distributed in possibly more than one store table. Therefore, mapping compilation component generates one view for each entity in CDL (levels, parent-child

relationships and fact relationships). Moreover, each SDL view is a query which combines store tables in the form of a view with the related CDL attributes.

4.1 SDL Views

The impedance mismatch between conceptual model and store model in CIM is managed by specifying correspondences. Each correspondence associates an attribute of a level in CDL to a set of column tables. The correspondences can also be expressed in the form of "Entity SQL", the data manipulative language that is used in EDM to express mappings and bidirectional views [57]. Entity SQL is a derivative of SQL designed to query and manipulate instances. In EDM, the mapping is represented in terms of queries on the entity schema of conceptual model and relational model. Here, correspondences can be expressed with the same structure, where each property mapping is a constraint of the form $Q_{entities} = Q_{tables}$. $Q_{entities}$ and Q_{tables} are queries over the conceptual schema and data store schema, respectively. These queries are written using an object-based approach, where levels and tables are treated as types and objects can be defined.

Example 4.1.1 *In Figure 3.2(c), the property mapping defined between* stateID *attribute of* State *and* state-province *column of* Branch *table can be represented as follows:*

SELECT *s.stateID*

FROM *State s*

=

SELECT *b.state-province*

FROM *SBranch b*

WHERE *b.isProvince="false"*

A property mapping describes how a portion of level instances corresponds to a portion of warehouse data. The mapping denoted in Example 4.1.1 describes that the element set of *stateID* attribute in *State* level is identical to the set of *state-province* values retrieved from *SBranch* table and restricted with "false" value of *isProvince* column. In case of having disjunctive correspondences, a correspondence consists of more than one mapping. Therefore, the elements assigned to the attribute with disjunctive correspondence, are the union of the data returned by each related mapping. It should be born in mind that a mapping does not need to provide a complete transformation that assembles a level from tables or vice versa, as it is in views. In CIM, views are generated such that they specify a complete structure of data for all constructs of conceptual model, not specifically levels.

It is not practical for a high-level user accustomed to only the conceptual view of data to come up with complex mapping definitions. However, the simple mappings defined by users convey neither the relation among attributes of a conceptual construct nor the columns in store tables. Our solution requires the user to only provide very simple property mappings between attributes in both models, which are later transparently compiled by the system into complex, fully-fledged, multidimensional mappings that can be used for query evaluation.

CIM conceptual and store models are related by *mappings*, called "SDL views", similar to those in data integration [45] or data exchange [41]. In particular, CIM uses mappings of the common form $c \rightsquigarrow \psi_s$, where c is an element in the *target* schema and ψ_s is a query (or view) over the *source* schema. In CIM, the conceptual model (CDL) functions as the target schema and the store model (SDL) as the source schema. The element c in the mapping expression is either a level, a parent-child relationship, or a fact relationship, whereas ψ_s is a view over the data warehouse's multidimensional tables.

CIM uses *sound* mappings, i.e., those defined by $\forall\bar{x}(\psi_s(\bar{x}) \rightarrow c(\bar{x}))$ expressions, where $\bar{x}$ denotes a set of variables.

Creating complex $c \rightsquigarrow \psi_s$ mappings requires defining a view ψ_s over the SDL for every level, parent-child relationship and fact relationship in the CDL — the *SDL Views*.

The following examples illustrate some of the SDL views for constructs in Figure 2.14 and 3.2. We assume a data warehouse in which all hierarchical relationships between dimension tables are captured by foreign key constraints.

Example 4.1.2 *The parent-child relationship between State and Country is represented in the SDL by a foreign key constraint from* SBranch *table to* SStateProvince *table. Based on the table definitions and such constraints, the view for the* State-Country *parent-child relationship is defined as follows:*

SELECT *SBranch.region* **AS** *regionID, SRegion.name* **AS** *name*
FROM *SBranch, SRegion*
WHERE *SBranch.region=SRegion.regionID*

The **SELECT** *clause contains only the attributes of* Region *constructs as they are mapped to the columns in SDL — in this case the* regionID *and name to* region *and name in* SBranch *and SRegion tables. The* **WHERE** *clause contains the foreign key (e.g., from region* foreign-key *element to regionID).*

As previously mentioned, it is possible that an attribute in a single CDL entity is mapped to attributes in multiple SDL tables. The next example illustrates this kind of disjunctive correspondence.

Example 4.1.3 *The* cityID *attribute in the CDL City is mapped to the* cityID *attribute in both* SCity *and* STown *tables. The view for the CDL* City *on the SDL is the following:*

SELECT *SBranch.cityID* **AS** *cityID, STown.name* **AS** *name,*
sk(SBranch.cityID, STown.name) **AS** *description*
FROM *SBranch, STown*
WHERE *SBranch.cityID=STown.cityID*

UNION

SELECT *SBranch.cityID* **AS** *cityID, SCity.name* **AS** *name,*
sk(SBranch.cityID, SCity.name) **AS** *description*
FROM *SBranch, SCity*
WHERE *SBranch.cityID=SCity.cityID*

The key attribute cityID *of the CDL* City *level corresponds to a column with the same name in the SVL SBranch table, while the other descriptive attribute (*name*) appears in both* SCity *and* STown *tables. These tables are connected to* SBranch *table by foreign keys. Since no correspondence has been specified for the* description *attribute, its value is computed by using a Skolem function (*sk *in the view above).*

CDL properties can be defined as *nullable* by setting its nullable XML attribute to true. The following example discusses view generation with nullable attributes in more details.

Example 4.1.4 *Assume that in Figure 2.14, the* name *attribute of the level* Region *is a nullable attribute of this level and the attribute mappings are defined as depicted in Figure 3.2. An outline of the possible view is the following:*

SELECT *SBranch.region* **AS** *regionID, SRegion.name* **AS** *name*
FROM *SBranch, SRegion*
WHERE *SBranch.region=SRegion.regionID*

With the above definition, which involves an inner join between SRegion *and* SBranch *tables,* Region *will contain only those regions that have non-null names. In contrast, using a left outer join between* SRegion *and* SBranch *tables retrieves appropriate data for the* Region *level.*

In the CIM framework, we assume all CDL attributes are non-nullable. Therefore, the issue of considering different join paths or different join operators is not addressed in the first version of the algorithm.

It is interesting to observe that the logical representation of a sound view that associates conceptual construct c to query q_s on the data warehouse, in this setting is as follows:

$$\forall x, q_s(x) \rightarrow c(x)$$

In other words, for each view the instances of c is a subset of instances generated by q_s.

This type of view discovery, used in our mapping compilation, is reminiscent of Clio [22]. However, our framework differs from Clio's in four key points: (i) each of our compiled mappings contain exactly all attributes of a single CDL schema construct, whereas Clio's mappings may contain only a subset of them, (ii) Clio views are select-project-join, while our views also include union in order to support disjunctive correspondences, (iii) our compiled SDL views use the minimum number of tables in the source schema, which is not a necessary condition in creating Clio mappings, and (iv) we support value conditions on correspondences (specified in the MDL mapping fragments) while Clio does not.

Example 4.1.5 *Consider the* City *and* Branch *levels; mapping compilation creates three separate views — one for City, one for Branch, and one for the City-Branch parent-child*

relationship between them. Each view contains attributes of the corresponding construct in CDL, and associates those attributes with a set of columns in SDL. The view in Example 4.1.3 is the union view which is compiled to compute the City *level with the minimum number of necessary joins. In contrast, Clio would only create two mappings — one for the join of SBranch and SCity, and the other for the join between SBranch and STown.*

4.2 Mapping Compilation Approach

In this section, the proposed algorithm for mapping compilation is explained intuitively. The compiled SDL views are expressed in SQL. The following definitions describe the components needed to build the SDL views. To begin with, we characterize the set of tables that appear in the **FROM** clause of the view definition. This is captured by the notion of a *join set*:

Definition 4.2.1 *A* **join set** $J = \{j_1, \ldots, j_n\}$ *over SDL schema* S *is a set where each element* $j_i \in J$ *is a table of* S*. A join set is* **well-formed** *iff for every pair* (j_k, j_l) *with* $j_k, j_l \in J, k \neq l$*, there exists some undirected path of referential constraints between* j_k *and* j_l*.*

Example 4.2.1 *It is easy to see that tables* SSale, SBranch, SCity *and* SClient *define a well-formed join set: every pair of tables in this set are connected by a chain of foreign key constraints (dashed arrows on the right side of Figure 2.14).*

Next we define the conditions in the **WHERE** clause of the compiled view. Each condition is given by a *join constraint* over the join set that defines the **FROM** clause:

Definition 4.2.2 *Let $J = \{j_1, \ldots, j_n\}$ be a join set of an SDL schema. A* **constraint** *over the join set J is an expression of the form* a $\langle$op$\rangle$ b*, where $\langle$op$\rangle$ is equality or inequality, and* a *and* b *are constants or columns of tables in J.*

The resulting SQL view will join the tables in a join set using the foreign keys and value conditions specified in the respective set of constraints (including those specified by users in MDL mapping fragments). A join set together with its join constraints define an *association*:

Definition 4.2.3 *An* **association** *is a pair $\langle J, C\rangle$, where* J *is a join set and* C *is a set of constraints over* J *such that for every referential constraint between tables in* J *there is a constraint in* C*.*

Example 4.2.2 *Consider the join set* J=$\{$*SSale, SBranch, SRegion*$\}$ *and the set of constraints* C =$\{$*SSale.branchID = SBranch.branchID, SBranch.region = SRegion.regionID, SBranch.isProvince = 'true'*$\}$*. The SQL query fragment corresponding to association $\langle J, C\rangle$ is the following:*

FROM *SSale, SBranch, SRegion*
WHERE *SSale.branchID = SBranch.branchID*
AND *SBranch.region = SRegion.regionID*
AND *SBranch.isProvince = 'true'*

The **FROM** *clause contains the members of join set* J *and the* **WHERE** *clause is a conjunction of the constraints in* C*. The constraint part consists of the foreign key constraints linking the tables in* J *together with an extra condition defined in one of the correspondences (e.g. those tuples of* SBranch *whose corresponding* isProvince *attribute is* true*).*

The *covering association* associates the tables and conditions in the mapping fragments:

Definition 4.2.4 *An association* $\langle J, C \rangle$ *is* **covering** *with respect to a set* M *of MDL mapping fragments iff* J *is a well-formed join set and for each mapping fragment* $m_i \in M$ *the following conditions are satisfied: (1) the SDL table of* m_i *is in* J*, (2) all conditions in* m_i *appear as constraints in* C*.*

Example 4.2.3 *Consider the correspondences linking* Region.regionID *and* Region.description *to* SBranch.region *and* SRegion.name*, respectively. These correspondences result in one MDL mapping fragment between* Region *and* SBranch*, and another between* Region *and* SRegion*, without conditions. The association* $\langle J, C \rangle$ *in Example 4.2.2 is covering with respect to these two mapping fragments because tables* SBranch *and* SRegion *belong to the join set* J*.*

Definition 4.2.5 *A covering association* $\langle J, C \rangle$ *w.r.t. a set of MDL mapping fragments* M *is* **minimal** *iff it is impossible to define a covering association* $\langle \mathrm{J}\prime, C \rangle$ *w.r.t.* M *where* J′ *has smaller cardinality than* J*.*

Example 4.2.4 *The covering association* $\langle J, C \rangle$ *in Example 4.2.2 is not minimal. The minimal covering association w.r.t. the mapping fragments in Example 4.2.3 is* $\langle J\prime, C\prime \rangle$*, where* $J\prime = \{SBranch, SRegion\}$ *and* $C\prime = \{SBranch.region = SRegion.regionID\}$*. The SQL fragment for* $\langle J\prime, C\prime\prime \rangle$ *is:*

FROM *SBranch, SRegion*

WHERE *SBranch.region = SRegion.regionID*

This covering association's minimality can be inferred from the fact that the regionID *and* description *attributes of* Region *are mapped to separate tables. Therefore, the minimum number of tables that can appear in its join set is two.*

Minimal covering associations minimize the number of joins that appear in the compiled view definitions.

Example 4.2.5 *Consider the following view for CDL Region level. It selects the* SBranch.region *and* SRegion.name *columns as* regionID *and name attributes of Region level from the join result of* SBranch *and* SRegion *tables. The* **FROM** *and* **WHERE** *parts are the minimal covering association from Example 4.2.4.*

SELECT *SBranch.region* **AS** *regionID, SRegion.name* **AS** *name*
FROM *SBranch, SRegion*
WHERE *SBranch.region = SRegion.regionID*

Compiling an SDL view v for a given CDL entity c requires two steps: (1) identify the mapping fragments in which c participates, (2) compute the minimal covering association w.r.t. those mapping fragments. We next describe the second step in detail.

4.2.1 Computing Minimal Covering Associations

Computing the minimal covering association uses a graphical model of the SDL schema that we call a *schema graph*:

Definition 4.2.6 *Given SDL schema S and graph $G = (N, E)$ where N is a set of nodes and E is a set of directed edges, let $\chi(s, g), s \in S, g \in G$ be a correspondence function between elements in S and G. G is a* **schema graph** *if $\forall n \in N, \exists$ table t in S s.t. $\chi(t, n)$ and $\forall$ edges $e = \langle n, n' \rangle \in E$, $\exists$ tables $t, t' \in S$ with referential integrity constraint c from t to t' s.t. $\chi(c, e)$. A schema graph is* **complete** *if for every table t in S, $\exists$ a node n in N s.t. $\chi(t, n)$ and for every referential constraint c in S there is an edge e in E s.t. $\chi(c, e)$.*

Example 4.2.6 *Figure 4.1(a) shows the complete schema graph of the SVL schema in Figure 2.14. For instance,* SBranch *is connected via a foreign key constraint to* SRegion*,* SCity*,* SStateProvince*,* STown *and* SSale *and therefore there is an edge between SBranch and these other tables.*

The SDL does not impose any condition on how dimensions and fact tables are connected. However, data warehouses are not arbitrary sets of tables and referential integrity constraints. In practice, fact and dimension tables follow certain patterns [46, 54]. Such schema patterns range from one fact table and one table per dimension (star schema) to multiple normalized dimension tables (snowflake schema) and multiple fact tables sharing dimension tables (constellation). Hybrids in which data has been partially normalized (starflake schemas) are also common. The notion of a *well-formed* SDL schema, formalized in Section 3.5, captures all the patterns mentioned above.

Computing a minimal covering association w.r.t. to a set of mapping fragments $f_1, \ldots, f_n$ begins by finding the well-formed join set for the association. First, we find the set of SDL tables $t_1, \ldots, t_m$ in $f_1, \ldots, f_n$. These tables might not define a well-formed join set: $t_1, \ldots, t_m$ are not necessarily pairwise connected via paths of referential constraints. In that case, a covering association must include additional "connecting" tables, as shown in the following example.

Example 4.2.7 *Suppose that CDL Client in Figure 2.14 has two additional attributes,* country *and* city*, that are mapped to* SStateProvince.country *and* SCity.cityID*, respectively.* Client *would participate in three mapping fragments:* Client-SClient*,* Client-SCity *and* Client-SStateProvince*. The SDL tables in those mappings fragments are* SClient*,* SCity *and* SStateProvince*. However, they do not define a well-formed join set because they are disconnected, i.e., there is no path of referential constraints between them. Thus, a covering association w.r.t. these mapping fragments must involve other tables. For in-*

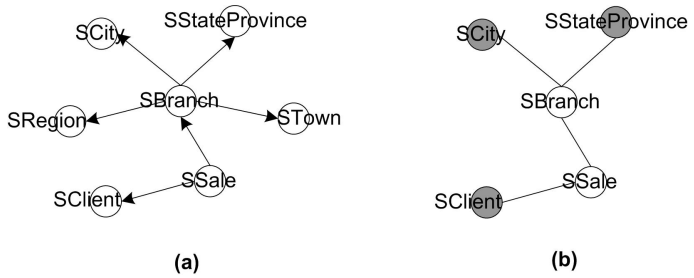

Figure 4.1: (a) Complete SDL schema graph for Figure 2.14 (b) Join tree for a covering association

stance, adding SBranch *and* SSale *to the join set makes it well-formed and therefore able to participate in a covering association. Figure 4.1(b) shows the five nodes defining a well-formed join set. The three painted black nodes correspond to SDL tables in the mapping fragments; the other two are the connecting nodes to make the join set well-formed. This is the join set of the minimal covering association w.r.t. the mapping fragments above.*

The compilation algorithm solves the issue of having well-formed join sets in two steps: (1) finding the shortest paths between all possible pairs of marked nodes, (2) finding the optimal way of connecting marked nodes together. In step (1), as some of the nodes might not be directly connected, the shortest distance between all marked nodes are computed using Dijkstra's algorithm [17], and the resulting paths are stored for later use. In case of the disconnection of two marked nodes, the path is empty and the path weight is assumed as MAXINT. The output of the first step is a *compressed graph*, i.e., a complete graph whose nodes are marked nodes of the store graph G and the edges are labeled by the length of the shortest path between adjacent nodes. In fact, each edge in a compressed graph represents a path in G and its weight is the length of such path.

Recall that the minimal covering association has a join set with the smallest possible cardinality. How do we obtain a well-formed join set that is minimal? Answering that question requires a few additional notions.

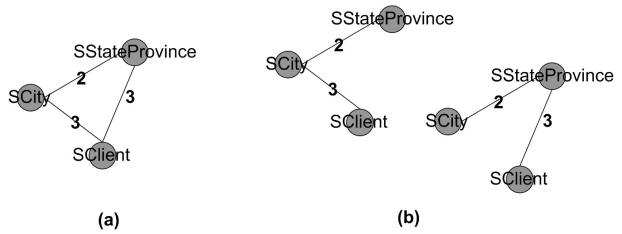

Figure 4.2: (a) Compressed graph; (b) Its two minimal spanning trees

Definition 4.2.7 *A **compressed graph** $C(G, M) = (N_C,\ E_C,\ w)$ of a schema graph G and a set of mapping fragments M is a weighted graph where N_C is the set of nodes, E_C is the set of edges, and w is a weight function that assigns an integer to each edge e in E_C such that: (1) every node n in N corresponds to a table t in M; (2) there is an edge $e = (n, n')$ in E_C with $w(e) = k$ iff the length of the shortest undirected path between n and n' in G is k.*

A compressed graph condenses the complete schema graph into the information needed to find a minimal covering association w.r.t. to a set of mapping fragments. We use *Dijkstra*'s algorithm [17] to compute the shortest paths between pairs of nodes. The length of the shortest path is the weight of the edge connecting two nodes. In case there are more than one shortest paths between two nodes, the weight of the edge between them is the sum of all the path lengths.

Example 4.2.8 *Figure 4.2(a) shows compressed graph $C(G,\ M)$ where G is the complete schema graph of Figure 4.1(a) and M is the set of mapping fragments of Example 4.2.7.*

We use the well-known *minimum spanning tree* algorithm [17] to find a subgraph of the compressed graph that connects all the nodes so that the sum of the weight of its edges is minimal. A spanning tree of a graph is a tree that connects all the graph nodes together. Since the weight of a tree is the sum of the weights of all its edges, a minimum

spanning tree is a spanning tree that weight less than or equal to the weight of every other spanning tree of that graph.

Once we compute the minimum spanning tree of the compressed graph, we must "expand" the edges by replacing them by the paths and nodes they represent. This expansion yields a schema graph whose nodes are the join set of the minimal covering association we wanted to compute. An overview of the mapping compilation algorithm is provided in Section 4.2.2.

Example 4.2.9 *Figure 4.2(b) shows the two possible minimum spanning trees of the compressed graph in (a). The expansion shows the node between* SCity *and* SStateProvince *(i.e.,* SBranch*) and the two nodes between either* SCity *and* SClient *or between* SCity *and* SStateProvince *(i.e.,* SBranch *and* SSale*). Once expanded, both spanning trees will produce the same schema graph: the one that appears in Figure 4.1(b).*

Finding paths in an SDL schema graph resembles following a chase in a database [51]. Our mapping compilation algorithm finds a subset of chases that span a minimum set of store schema tables. In general, the chase may not terminate, and computing a chase can itself be exponential in the worst case [36]. However, the join paths that we use are restricted by the well-formed structure of a data warehouse (Definition 4.2.1). This restriction guarantees the termination and polynomial time complexity of our algorithm, even in the worst case.

4.2.2 Mapping Compilation Algorithm

The mapping compilation algorithm gets as input CDL, SDL and MDL files. It extracts the schema of the conceptual and store models as well as correspondences by parsing these XML files. The SDL views are generated, as SQL queries in string format, according

to the obtained information from schemas and mappings. The type of view that is generated for a level has all attributes of the level as its columns. However, a parent-child relationship consists of two columns which are the key attributes of the parent level and child level. By associating the instances of key attributes of two levels, the other attributes of the levels become semantically related. This way, we intend to instantiate the roll-up function from the child level to parent level by enumerating the instances. In case of the fact relationships, the generated views consist of the attributes of the fact relationship and the key attributes of dimensions' bottom levels as its columns. In other words, the view of a fact relationship demonstrates a data cube for the measures and the most granular dimension data.

Example 4.2.10 *The mapping compiler generates the following SDL view for the level* Region*:*

SELECT *SBranch.region* **AS** *regionID, SRegion.name* **AS** *description*
FROM *SBranch, SRegion*
WHERE *SBranch.region=SRegion.regionID*

The columns of this view are the attributes of Region *level that can be found in Figure 2.14. The* FROM *and* WHERE *parts of this view are generated according to the minimal covering association with respect to the arrows between* regionID *and* description *in* Region *level and* region *and* name *in* SBranch *and* SRegion *tables.*

Example 4.2.11 *The SDL view of Branch-Region relationship is defined as follows:*

SELECT *SBranch.branchID* **AS** *branchID, SBranch.region* **AS** *regionID*
FROM *SBranch*

The attributes branchID *and* regionID *are respectively key attributes of* Branch *and* Region *levels. They are defined as the columns of* Branch-Region *view. The correspondences that are involved in creating the view are the ones that associate* branchID *and* regionID *to* branchID *and* region *columns of* SBranch *table.*

Example 4.2.12 *The SDL view that is generated for fact relationship* Sale *is as follows:*

SELECT *SBranch.branchID* **AS** *branchID, SClient.clientID* **AS** *clientID,*
SSale.staffCommission **AS** *staffCommission, SSale.totalRevenue* **AS** *totalRevenue,*
FROM *SBranch, SSale, SClient*
WHERE *SBranch.branchID=SSale.branchID*
AND *SClient.clientID=SSale.clientID*

This view is a cube in which staffCommission *and* totalRevenue *are measures and take their elements from the columns with the same name in* SSale *table. The descriptive data of these measures are specified in* branchID *and* clientID *columns of the view which are associated to columns in* SBranch *and* SClient *tables, based on the mappings defined in Figure 2.14 for these two attributes. The foreign keys between* SBranch, SClient *and* SSale *tables, which are described as join conditions in this view, express the relation between dimensions and fact data.*

Figure 4.3 shows the mapping compilation algorithm. Line (2) creates a complete schema graph from the SDL; the schema graph is generated once and used through the entire compilation process. For each schema element in the CDL model (levels, parent-child relationships and fact relationships) we create one view (lines (3-17)), which identifies the instances of that element. We first find the mapping fragments relevant to the current CDL schema element (line (4)). By relevant mapping fragments of an element, we mean those which contain the property mappings of the element. Then

```
COMPILE_MAPPINGS(S,C,M)
Input: SDL schema S, CDL schema C, and MDL model M
Output: The set of compiled views VS over S
(1) VS ← 0;
(2) SG ← CREATE_SCHEMA_GRAPH(S);
(3) foreach schema element X of C
(4)    MF ← GET_MAPPING_FRAGMENTS(C,M,X);
(5)    CG ← CREATE_COMPRESSED_GRAPH(SG,MF);
(6)    ST ← COMPUTE_MST(CG);
(7)    XT ← EXPAND_MST(ST);
(8)    A  ← DEFINE_MCA(XT,MF);
(9)    if A = ∅;
(10)       return ∅; "Inconsistent Correspondences"
(11)   if X is a level
(12)       VS ← VS ∪ COMPILE_LEVEL(S,X,M,A);
(13)   if X is a parent-child relationship
(14)       VS ← VS ∪ COMPILE_PC_REL(S,X,M,A);
(15)   if X is a fact relationship
(16)       VS ← VS ∪ COMPILE_FACT_REL(S,X,M,A);
(17) return VS;
```

Figure 4.3: Compile Mappings Algorithm

we compute the minimal covering association for the mapping fragments. This involves creating the compressed graph (line (5)), computing the minimum spanning tree (line (6)) and expanding such a tree (line (7)) to obtain the schema graph XT whose nodes are the join set J of the minimal covering association A for the view. The set of constraints C of A is obtained by putting together the join conditions in XT and the conditions from the mapping fragments MF used (line (8)). The final process consists of creating the SQL query of the view by invoking one of three different algorithms depending on the type of element: if the CDL schema element is a level (COMPILE_LEVEL in line (12)), if it is a parent-child relationship (COMPILE_PC_REL in line (14)) or if it is a fact relationship (COMPILE_FACT_REL in line (16)).

The function GET_MAPPING_FRAGMENTS(C,M,X), invoked in line (4) of function COMPILE_MAPPINGS, scans the MDL file to find the mapping fragments which associate the element X to tables in underlying data warehouse. MDL illustrates the property mappings between level attributes and table columns. It is straightforward to find the relevant

```
GET_MAPPING_FRAGMENTS(C,M,X)
Input: CDL schema C, MDL schema M, element X in C
Output: The set of mapping fragments for X
(1) if X is a level;
(2)    return all mapping fragments p_i ∈ M containing X;
(3) if X is a parent-child relationship
(4)    MP ← all mapping fragments p_i ∈ M containing
           the parent level in X;
(5)    MC ← all mapping fragments p_i ∈ M containing
           the child level in X;
(6) return MP ∪ MC;
(7) if X is a fact relationship
(8)    MC ← all mapping fragments p_i ∈ M containing X;
(9)    foreach dimension D_i referenced in X
(10)       MC ← MC ∪ mapping fragments p_i ∈ M containing
               the bottom level of D_i
(11) return MC;
```

Figure 4.4: Get Mapping Fragment Algorithm

property mappings to X, if it is a level. This can be done by finding those fragments in MDL which have the name of X as the value of their *level* attribute. However, there is not any explicit mapping between parent-child relationships and data warehouse tables. Since, the parent-child relationship views contain the key attributes of related parent and child levels, the relevant mapping fragments are those with property mappings of parent and child key attributes, which are labeled by "key" inside "level" element in CDL. Likewise, the meaningful mapping fragments of a fact relationship are the ones that describe the property mappings of bottom levels' key attributes and fact relationship attributes, which are expressed by "role" label in "factRelationship" element inside CDL.

In our framework, users are capable of mapping an attribute to more than one columns by defining disjunctive correspondences. It is inferred that an attribute with disjunctive correspondence takes its instances from multiple columns. An SDL view integrates the related instances of all attributes of a conceptual element into one unit. The relation of columns that feed the instances of conceptual attributes is described by joining the tables to which they belong. Therefore, if an attribute is mapped to more than one columns, the resulting instances should contain all combinations of columns to which

attributes of the element were mapped. We create a subquery for each combination and then union all of them. For instance, in mapping the attributes of *City* level to *STown* and *SCity* tables, *cityID* and *name* both are mapped to two columns. In order to find the *City* SDL view, we generate four compressed graphs; in each graph, one combination of property mappings for *cityID* and *name* attributes is considered for marking nodes. For example, the combination of the property mapping from *cityID* in *City* to *cityID* in *SCity* and *name* in *City* to *name* in *STown* results in marking two nodes *STown* and *SCity* in the graph schema and having two nodes in the relevant compressed graph. Considering all combinations, four different compressed graphs are built to have four minimal covering associations. These minimal covering associations are translated into subqueries which are later unioned as one SDL view query for *City*. As discussed, when the correspondences are disjunctive, various combinations of correspondences, thus more than one association, should be taken into account (As in line (1) of COMPILE_LEVEL function). In such cases, a view is created for each association and the final view is the **UNION** of these views, as done in lines (9-11) of COMPILE_LEVEL. For instance, if a construct has m attributes, each mapped to e_i, $1 \leq i < m$ columns, the related SDL views consist of $e_1 \times \ldots \times e_m$ query sections.

These three functions create the actual SQL expression of the views by compiling the subexpressions first and then putting them together in one single expression using **UNION**. In the COMPILE_LEVEL algorithm, this is achieved, firstly, by including the *column.table* combinations in the **SELECT** part of the view (line (2)). These *column.table* combinations are specified in the mapping fragment as the table and column to which a CDL attribute is mapped. The keyword "**AS**" in line (2) of COMPILE_LEVEL function specifies which column of an SDL table is mapped. As we have discussed, users might also put some selection restrictions on the mapping fragments. For instance, in Figure 2.14,

only some of the values of state-province column are considered as Province instances, according to isProvince column value of the same table. Thus, the join set constraint W of association A_i should be augmented by the conditions specified in the related mapping fragments, as in lines (5-6) of COMPILE_LEVEL algorithm. A view is formed in line (7) by assembling the column.table fragments as the **SELECT** clause, the join set of the association as the **FROM** clause, and the join set constraint of the association and additional conditions as the **WHERE** clause. In case of having more than one minimal covering association, a view is created for each association and the final view is the **UNION** of these views, as done in lines (9-11) of COMPILE_LEVEL.

In the COMPILE_PC_REL algorithm, views are assigned to parent-child relationships in order to specify the respective roll up function between children and parents. We add the views that have been already generated for parent and child levels to the join set of the association A_i (lines 1-2). It is possible that the join set contains the tables appearing in the views. In that case, the tuples returned by the parent-child relationship view are already filtered and the views are not added to the join set. If views are added to the join set of A_i, the **WHERE** clause of the view should describe how the views are joined with other tables. The **SELECT** clause of such views is the columns to which these key attributes are mapped (line (4)). The **WHERE** clause is composed of the join set and possible user defined constraints (lines (7-9)).

In lines (10-15), the **WHERE** clause W of the query specifies that the join between the parent/child views and other tables in the minimal covering association is defined on the key attributes of parent/child levels and the columns of the tables (in join set of the association) to which such attributes are mapped. The **FROM** part, F, of a parent-child view is the join set of the minimal covering association.

```
COMPILE_LEVEL(S,C,M,A_s)

Local: string query V;
       query set U;
       string set T;
       string MP;
(1) foreach association A_i in A_s
(2)     let T be all strings:
        column.table+" AS "+X.property fragment
        built based on the corresponding set of A_i;
(3)     let F be the join set part of association A_i;
(4)     let W be the join set constraint of association A_i;
(5)     foreach m_i covered by A_i
(6)         W ← W ∪ the conditions of m_i described
                in MDL
(7)     let MP be the string:
                "SELECT "+T+ "FROM "+F+"WHERE "+W;
(8)         U ← U ∪ {MP};
(9) V ← the first member of U;
(10) foreach MP in U other than the first one
(11)    append "UNION "+MP to V;
(12) return V;
```

Figure 4.5: Algorithm for Level View Compilation

4.2.3 Soundness of Generated SDL Views

The views we generate are always *sound*, for all data warehouses. By soundness, we mean that the data returned by views do not contain any element that should not be extracted from data warehouse according to the semantics of property mappings provided by the user. We need the views to be lossless, meaning that the data is put in the store model can be completely retrieved in the generated views. The proposed mapping compilation algorithm is based upon the well-known chase problem [22]. The chase describes a way to combine the data stored in different relations/tables and find the related portion of it. In order to find a chase, we might be forced to consider all possible paths, which makes a chase not to terminate in some cases. However, instead of taking all paths, we select some of them, according to our minimality criterion, and do the join operation based on the foreign keys that are a part of the definition of the related chase. In some

```
COMPILE_PC_REL(S,X,M,A_s)

Local: string query V, V_p, V_c;
       query set U;
       string set T;
       string MP;
(1) let V_p be the view generated for the parent
    level in X;
(2) let V_c be the view generated for the child
    level in X;
(3) foreach association A_i in A_s
(4)     let T be all strings:
            column.table+" AS "+X.property fragment
            built based on the corresponding set of A_i;
(5)     let F be the join set part of association A_i;
(6)     F ← F ∪ V_p, V_c;
(7)     let W be the join set constraint of association A_i;
(8)     foreach m_i covered by A_i
(9)         W ← W ∪ the conditions of m_i described
            in MDL;
(10)    foreach m_i, covered by A_i, describing
                the key attribute k_i of the parent level of X
(11)            W ← W ∪ a condition on equality
                of V_p.k_i and column.table specified in m_i for k_i;
(12)    foreach m_i, covered by A_i, describing
            the key attribute k_i of the child of X
(13)        W ← W ∪ a string condition on equality
                of V_c.k_i and column.table specified in m_i for k_i;
(14)    let MP be the string:
                "SELECT "+T+ "FROM "+F+" WHERE "+W;
(15)        U ← U ∪ {MP};
(16) V ← the first member of U;
(17) foreach MP in U other than the first one
(18)    append UNION MP to V;
(19) return V;
```

Figure 4.6: Algorithm for Parent-child View Compilation

store schemas, it happens that there are more than one foreign key paths between two tables and all the data in one table does not associate to the data in the other one following only one of the paths. In this case, the relation between the data in the tables is extracted by joining the data along all paths and then unioning the data obtained for each path. However, if all data in one table participates in each of the possible paths (the definition of well-formed data warehouse), considering only one path will result in all data relations. In this case, our mapping compilation algorithm considers the shortest path in order to have the minimum number of joins. In conclusion, following the idea of

```
COMPILE_FACT_REL(S,X,M,A_s)

Local: string query V;
       A list of query sets V_i;
       query set U;
       string set T;
       string MP;
(1) foreach dimension D_i in the dimension set of X
(2)     let V_i be the corresponding view generated for
        the bottom level of D_i;
(3) let V be a view;
(4) U ← 0;
(5) foreach association A_i in A_s
(6)     let T be all strings:
            column.table+"AS"+X.property fragment
            built based on the corresponding set of A_i;
(7)     let F be the join set part of association A_i;
(8)     foreach view V_i
(9)         F ← F ∪ V_i;
(10)        let W be the join set constraint of association A_i;
(11)        foreach m_i in covered by A_i
(12)            W ← W ∪ the conditions of m_i described
                    in MDL;
(13)    foreach D_i in the dimension set of X
(14)        let k_i be the key attribute of D_i's bottom level;
(15)        W ← W ∪ a string condition on equality
                of V_i.k_i and column.table specified in m_i for k_i;
(16)    let MP be the string:
                "SELECT "+T+"FROM "+F+" WHERE "+W;
(17)        U ← U ∪ {MP};
(18) V ← the first member of U;
(19) foreach MP in U other than the first one
(20)    append UNION MP to V;
(21) return V;
```

Figure 4.7: Algorithm for Fact-Relationship View Compilation

writing views using minimal covering association (finding the shortest path), our views contain all the related data (exact) for well-formed data warehouses and a subset of related data (still sound) for non-well-formed data warehouses. When dealing with well-formed data warehouses, the generated views are *exact*. This is because well-formed data warehouses have total roll up functions. For instance, consider a *Date* dimension with total roll up functions where *Day* (bottom) rolls up to both *Month* and *Week*, *Month* rolls up to *Quarter*, and *Quarter* and *Week* both roll up to *Year* (top). In that case, we can safely join *Day*-*Week*-*Year* to compute a minimal covering association for *Day*

and *Year* because *Day-Month-Quarter-Year* is longer and yields exactly the same set of tuples. Our running example (Fig 2.14) has some partial roll up functions (from City to Province and State). Computing a minimal covering association for *City* and *Country* by joining either *City-Province-Country* or *City-State-Country* would create a view that is sound but not complete.

There is always the option of simply modifying the algorithm such that it finds all paths between two marked nodes and create a union of subqueries of the paths. However, since most common data warehouse schemas are well-formed (as discussed in Section 3.5), the proposed mapping compilation algorithm return exact *views*. However, in case of data warehouses with non-well-formed schemas, we compromise the exact views for the sake of having better views.

4.3 Computational Complexity of the Mapping Compilation Algorithm

The algorithm generates the SDL schema graph once for the whole compilation process. For a configuration with S store tables and F references between distinct tables, the time complexity of a store graph creation is $O(S+F)$. The algorithm reuses the shortest paths of already processed marked node pairs. This is done by storing all found shortest paths and computing a path only in case that at least one of the nodes are a new marked one. By storing the reference graph in the form of adjacency lists and using a binary heap to extract the minimum efficiently, finding the shortest path between each node pair requires $O(S \log F)$ time [17]. The worst case performance of Dijkstra's algorithm, however, is $O(F + S \log S)$. The shortest path subroutine is invoked by our algorithm for all distinct marked pairs. We assume C^2 number of distinct pairs, where C is the average number

of correspondences specified by the user for each construct in the CDL. In other words, C is the average number of marked nodes in each execution of this step. In the worst case C is of $O(S)$ complexity. Therefore the number of nodes labeled according to the correspondences and their pairs can be presumed linear size. Accepting this assumption, the complexity of running the first step of our algorithm is $O(C^2 \times F + C^2 \times S \log S)$ time.

The second step of each run is to find the minimum spanning tree of a compressed graph of average size C in terms of nodes. Currently, there are two commonly used algorithms: Prim's algorithm and Kruskal's algorithm [17]. Both of these algorithms are greedy in nature and run in polynomial time. For example, an implementation of Prim's algorithm that uses the binary heap data structure and the adjacency list representation, requires $O(C^2 \log C)$ running time for a complete graph of size C. Using the adjacency matrix graph representation and an array of weights to search, the algorithm can be shown to run in time $O(C^2)$. Unlike the first step, this step is performed only once in each run of the process of generating a view for each construct in the CDL. The mapping compilation is executed for all constructs in the CDL, including levels, parent-child relationships and fact relationships. Thus, begin L the number of CDL constructs, the algorithm spends $O(L \times C^2 \log C)$ time on this step. Consequently, the process of compiling all views in our framework is of $O(S + F) + O(C^2 \times F + C^2 \times S \log S) + O(L \times C^2 \log C)$.

Chapter 5

Experimental Evaluation

In Section 4.2.3, we discussed the correctness of the generated views by referring to chase problem. Our objective in this section is to show that the presented mapping algorithm is designed such that it can generate views for any conceptual, store and mapping schemas, with a practically good time complexity. In this respect, we present the results of applying mapping compilation on different conceptual model with different sizes, that are mapped to various store models through different mappings. The conceptual models are chosen from standard and benchmark models. Whilst, the store models vary from well-formed to random database schemas with different number of tables and different foreign key numbers and structures.

We test the impact of different factors on the efficiency of mapping compilation algorithm by running experiments on different benchmarks. According to the theoretical complexity of compiling mappings into views, discussed in Section 4.3, we can extract the involving factors in performance of the compilation process. In general, three main keys are considered: (1) the size of the conceptual model (in terms of the number of constructs), (2) the size and structure of store model (in terms of the number of tables and foreign keys as well as the connections between tables), and (3) the size and

complexity of correspondences (in terms of the number of different tables to which one level in conceptual model can be mapped to as well as the number tables to which each attribute is mapped). We chose three benchmarks as the conceptual models, which have various number of constructs: (1) TPC-H Benchmark [24], (2) the Dream Home case study [15] discussed in Section 3.1, and (3) Infection Control Data Warehouse (in use at the Ottawa Hospital) [21]). In order to study the other two factors related to store models and mappings, we implemented an SDL and MDL generator, that generates different store models, with various number of tables and diverse structure, for each conceptual model benchmark. Then, the generator maps each store model to the conceptual model, by defining correspondences of different size and type. Considering all cases, the views are created as the output of compilation. The compilation is evaluated for each case of a conceptual model mapped to a randomly-generated store model through correspondences.

5.1 Setup

We implemented our mapping compilation algorithm (Chapter 4) in Java. We used MySQL [1] for physical storage, Codehaus streaming API for XML, StAX [2] for parsing the XML files, and the Annas [3] graph and algorithm package — which includes an efficient implementation of the Dijkstra and Prime algorithms. Each SDL model is generated from the MySQL schemas of the tested physical models. The output of the mapping compilation algorithm is a text SQL query definition for the views created.

Our experiments explored the four parameters that impact our mapping compilation algorithm the most: (1) number of entities in the CDL, (2) number of mapping fragments

[1] http://www.mysql.com/
[2] http://stax.codehaus.org/
[3] http://code.google.com/p/annas/

Benchmark parameters	TPC-H	Dream Home	Infection Control
(1) CDL size	21	16	9
(2) MDL size (range)	55–330	51–376	96–596
(3) SDL size (range)	7–140	9–180	8–160
(4-a)# Foreign keys (range) –Fully-connected schema	21 – 9,730	28 – 16,110	79 – 12,720
(4-b)# Foreign keys (range) –Well-formed schema	5 – 1,667	5 – 1,990	46 – 4,433
(output) # Views generated	21	16	9

Table 5.1: Benchmark parameters

in the MDL, (3) number of tables in the SDL, and (4) "density" of referential constraints in the store schema (i.e., number of foreign keys between tables). We used three different conceptual models (parameter (1)). For each model, we varied parameters (2), (3) and (4) to study scalability.

The experiments were performed on three multidimensional schemas: the TPC-H Benchmark [24], the Dream Home case study [15] discussed in Section 3.1 and an Infection Control Data Warehouse (in use at the Ottawa Hospital) [21]. TPC-H is a well-known benchmark containing a schema and a suite of business-oriented queries. The Dream Home model was described in the Introduction. The Infection Control Data Warehouse is a conceptual model showing information about patients' admission, activities, and discharge in order to monitor and reduce the risk of infections.

To build store schemas of different sizes, for each run we generated a uniformly distributed random number of tables bounded by a given maximum number. All experiments were conducted on the same Pentium(R)D 3.40 GHz with 2 GB of RAM running Windows XP. The results are all averaged over 500 execution times in order to cover different SDL cases and MDL mappings.

In each run, we create a store schema for the underlying data warehouse of a benchmark conceptual model. The characteristics of this schema is set according to a set of parameters: maximum number of tables, structure of foreign keys and maximum number of tables in a disjunctive correspondence. First, a random set of tables are generated and

connected through foreign keys. The number of tables is a parameter which is changed in different runs. Tables refers to each other based on a given structure. For instance, we can connect them with a complete graph structure, that is, each table has foreign keys to all other tables. This way, the SDL schema has many foreign key references. However, in a different structure, we only define the foreign keys such that all tables are directly or indirectly connected. This results in testing the performance of mapping compilation in the presence of fewer number of foreign keys. One of the input parameters of this testing process is the maximum number of tables to which each level in conceptual model maps. In order to do the mapping between the benchmark conceptual model and created store model, for each level, we generate a random number (let's call it m_i) smaller than the input parameter. Then, m_i tables are selected from sore tables and mapped to the level. In order to test the effect of disjunctive correspondences we assume that attributes of levels can get their elements from a maximum number of tables. The maximum number is an input parameter which varies in different runs. The number of tables from which an attribute can be mapped to is also selected randomly; then, tables are selected from the SDL as the sources of data of this attribute. We chose the number of 500 runs such that sufficiently different store schemas and mappings are tested in the process of the described semi-randomly generation of SDL and MDL.

We tested two types of SDL schemas: *well-formed* and *fully-connected*. The well-formed schemas satisfy Definition 3.5.1. In the fully-connected schemas, each SDL table has foreign keys to all other tables in the schema. The fully-connected schemas are not well-formed; however, they allow us to study the performance of our algorithm in the presence of unusually high numbers of referential constraints. As previously discussed, the algorithm returns exact views only in the case of well-formed schemas. For the fully-connected schemas the views generated are sound. In all cases the algorithm generates

a view for each level, parent-child relationship and fact relationship of the conceptual schemas.

The first three rows of Table 5.1 respectively summarize the range of sizes of the CDL, MDL and SDL of each test domain. Size is measured by the number of levels, parent-child relationships and fact relationships for the CDL, mapping fragments for the MDL, and tables for the SDL. The fourth and fifth rows, report the number of foreign keys for the fully-connected and the well-formed schemas, respectively. The sixth row provides the number of views generated by the mapping compilation algorithm in each case.

5.2 Experimental Results

The first set of experiments evaluates the performance of the compilation algorithm for well-formed schemas (Figures 5.1, 5.2 and 5.3) with a fixed CDL schema and varying the number of SDL elements and mapping fragments. These figures depict the time to generating views for all constructs in each conceptual model versus the average number of SDL tables. We varied the number of mapping fragments to which level can be mapped from 1 to 7, e.g., for 7 mapping fragments per level, each level is mapped to an average of 7 different tables in the SDL.

Our second set of experiments shows the impact of the density of foreign key references among tables on compilation time. We consider the same store tables as in the previous experiments and again vary the number of SDL elements and mapping fragments. However, this time the SDL tables have a fully-connected topology, i.e., the schemas contain the maximum number of possible foreign keys among their tables. This worst case provides us with an approximation of the upper bound for the execution time. The results are shown in Figures 5.4, 5.5 and 5.6.

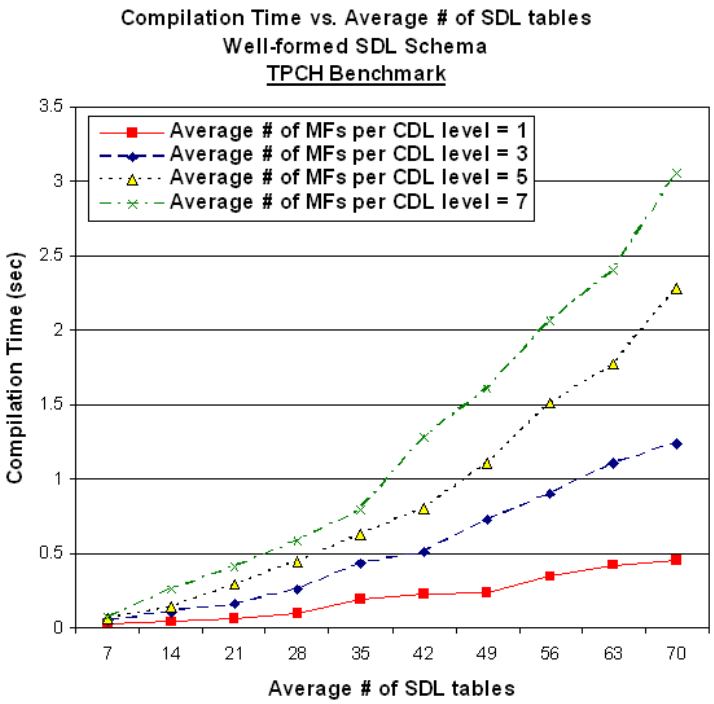

Figure 5.1: Compilation Time vs. Average # of SDL tables for well-formed SDL schemas - Well-formed SDL Schema for TPC-H Benchmark

Our results suggest that the performance of the mapping compilation algorithm in practice is better than its worst-case complexity. Consider the curve corresponding to 5 mapping fragments per CDL level in the TPC-H Benchmark with a well-formed schema (Figure 5.1). When the number of tables is increased by one order of magnitude, the average compilation time grows by just 15 times to a total of around 9 seconds (for 90 tables).

In practice most data warehouse schemas are either well-formed or easily transformed to well-formed using techniques in Section 4.2.1. Hence the first set of experiments fairly accurately represent mapping compilation in real data warehouses. Since most data warehouses usually have a few dozen tables and rarely over a hundred, our experiments cover an important range of real world data warehouse sizes.

In summary, in spite of a relatively high polynomial worst-case complexity, compila-

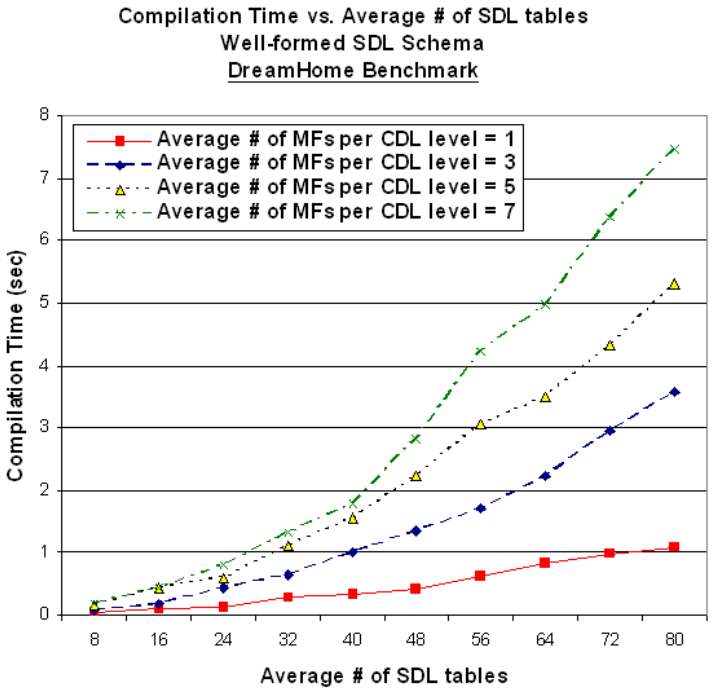

Figure 5.2: Compilation Time vs. Average # of SDL tables for well-formed SDL schemas - Well-formed SDL Schema for DreamHome Benchmark

tion is fast in practice: it only takes in the order of seconds on large well-formed store schemas. Even for the artificial worse case of the fully-connected graph, which does not appear in practice, compilation is on the order of minutes for large data warehouse schemas.

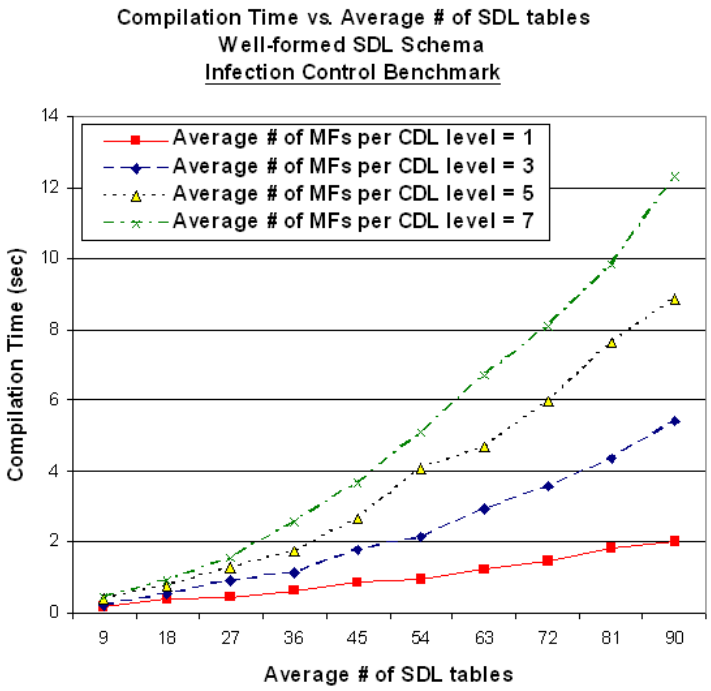

Figure 5.3: Compilation Time vs. Average # of SDL tables for well-formed SDL schemas - Well-formed SDL Schema for Infection Control Benchmark

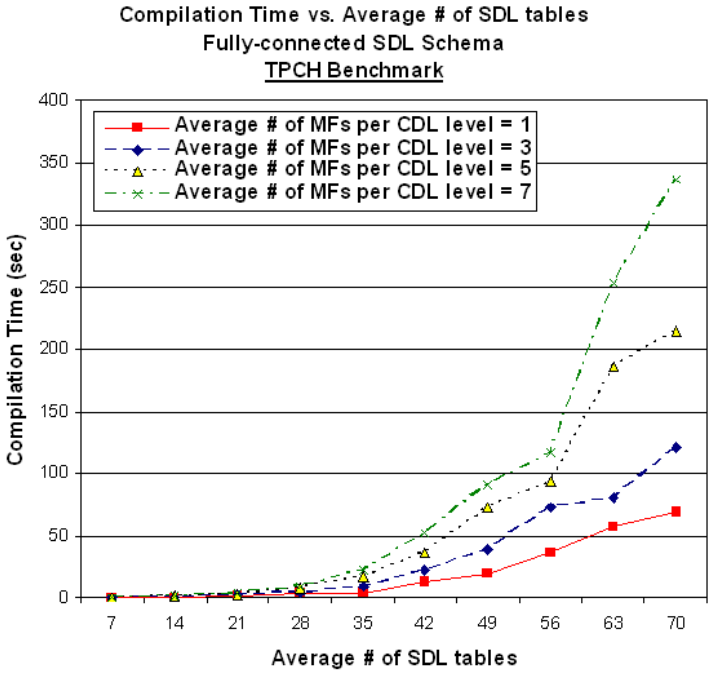

Figure 5.4: Compilation Time vs. Average # of SDL tables for well-formed SDL schemas - Fully-connected SDL Schema for TPC-H Benchmark

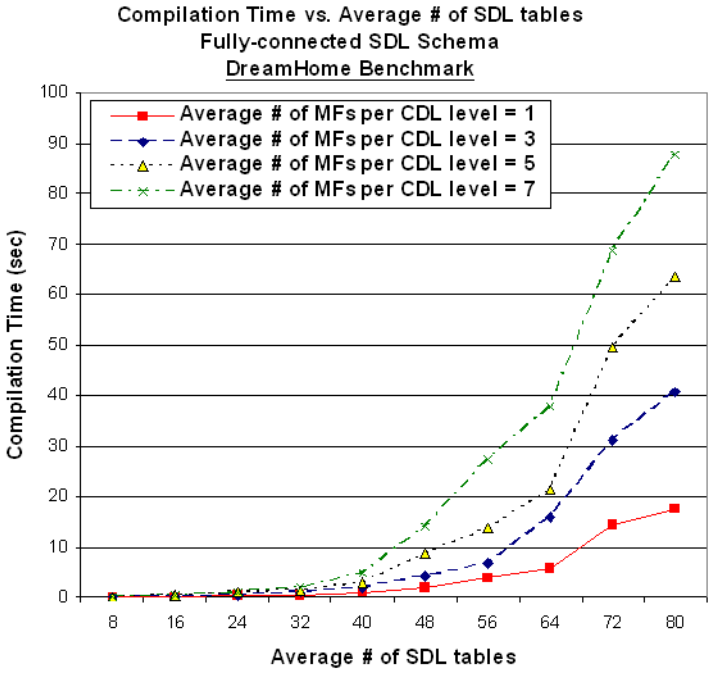

Figure 5.5: Compilation Time vs. Average # of SDL tables for well-formed SDL schemas - Fully-connected SDL Schema for DreamHome Benchmark

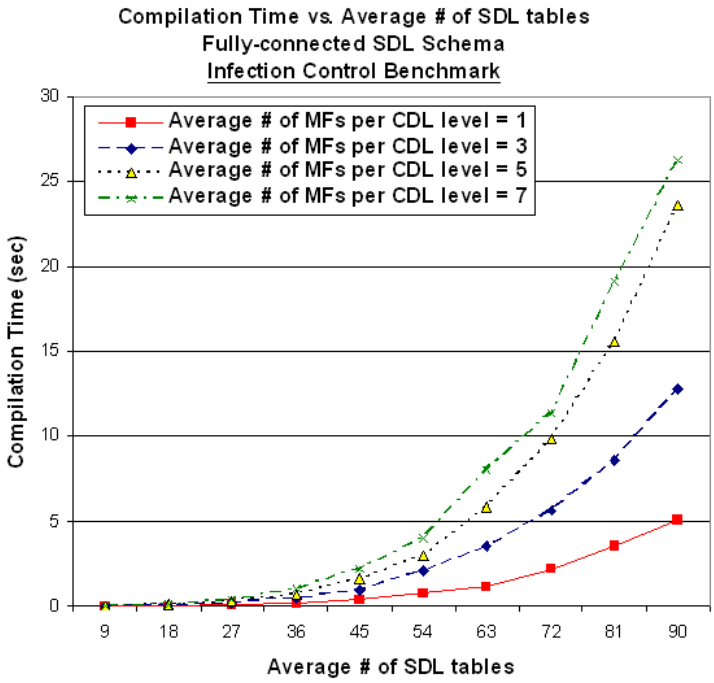

Figure 5.6: Compilation Time vs. Average # of SDL tables for well-formed SDL schemas - Fully-connected SDL Schema for Infection Control Benchmark

Chapter 6

Conclusion

6.1 Summary of Contributions

In Section 1.3 we outlined the key contributions of this research and in Section 3.7.3 and 4.2 we elaborated on these points. Here we discuss the impact of this work.

Contribution 1: An algorithm for refactoring non-summarizable dimension models and generating summarizable dimensions.

An algorithm for refactoring non-summarizable dimension models was proposed in Section 3.7.3. A dimension might be non-summarizable due to the heterogeneity of data instances. Summarizability of dimension models assists in easy and efficient aggregate navigation. We defined dimension summarizability concept and its condition in the context of CIM (Conceptual Integration Model) framework. By identifying this condition in our own setting, we were able to better understand and conceptualize a possible data-driven solution for making dimensions summarizable. We use *Bisimulation* algorithm [38] to split the data elements of dimension levels and organize them into a new set of levels, thus dimension schemas, such that they satisfy summarizability criterion. In other

words, for each non-summarizable dimension we come up with a set of new schemas and instances that convey the same dimensional model but in a summarizable format.

Contribution 2: A query answering mechanism for refactored summarizable dimensions.

In Section 3.7.5, we outlined a mechanism for answering queries against summarizable dimensions. The definition of summarizable dimension/hierarchy guided us to a simple query answering method, which was theoretically proved to be sound. The query answering engine integrates the answers of the queries posed against all dimensions/hierarchies that were generated by refactoring algorithm. In summary, if a dimension happens to be non-summarizable, we can rebuild its schema and data such that it becomes summarizable. Then, any query can be manipulated on the dimension successfully.

Contribution 3: A mapping compilation algorithm that translates the mappings, within CIM framework, between conceptual model and store model into a set of query views in terms of store tables.

We developed a mapping compilation algorithm for defining a set of views on top of the data warehouse. These views bridge the gap between the conceptual model and data warehouse schema. Each level, parent-child relationship and fact relationship has a view generated by the mapping compilation algorithm. The views associated to levels identify the data instances of a level, while a parent-child relationship view enumerates the relations between parent elements and child elements. In addition, the view generated for each fact relationship illustrates the base cube with attributes of the fact relationships as its measures and key attributes of dimensions' bottom levels as its dimensions. Obviously, there might be heterogeneity, thus non-summarizability, in the data returned by these views. The refactoring algorithm for solving summarizability solution can be applied on the data of the views in order to have a new summarizable dimension model.

This model can be used in answering queries that are posed against the conceptual model. The mapping compilation provides sound views with minimum number of joins among tables. A discussion on well-formedness of a data warehouse is done in order to study the characteristics of the generated views. Based on the definition of views, the generated views are exact (they return all the related data) when the data warehouse is well-formed. Most common structures for data warehouses satisfy well-formedness criterion. However, in case the data warehouse does not meet this condition, the views are sound (they return a correct portion of the related data). A solution is also proposed for making views in the presence of non-well-formed data warehouses. Additionally, we studied the complexity of the compilation process and proved it to be polynomial. The mapping compiler was implemented. We tested the compiler on three different benchmarks (one real-world (Infection Control) and two standard models (TPC-H and DreamHome)). The impact of the size and complexity of underlying data warehouse and density of mappings was evaluated on the mapping compilation efficiency. Based on the experimental results, we concluded that the algorithm works better than the expected polynomial time in practice. This is due to a consideration in implementation. We store the information extracted in the process of generating views and reuse it in other view generation runs. Therefore, an incremental information gathering during mapping compilation results in more efficient mapping compilation than what theoretically is expected.

6.2 Thesis Limitations

The views generated by mapping compilation are not *exact* for warehouses that are not well-formed. When dealing with well-formed data warehouses, the generated views are exact. In the proposed compilation algorithm, we support equality and inequality selection conditions on the mappings provided by users. However, there is potential

for users to map an attribute in conceptual model to a function of column(s) in data warehouse tables. Moreover, extracting join paths by considering the shortest paths is a heuristic. These join paths are not necessarily optimal. In practice, other factors, such as indices and the size of data stored in underlying database, also influence the optimality of join paths.

6.3 Future Work

We are considering extending this work in several directions, the first of which is extending mapping algorithm and the second is query evaluation.

6.3.1 Multidimensional Conceptual Query Evaluation

The views that are generated by mapping compilation virtually instantiate the constructs of CIM conceptual model (levels, parent-child relationships and fact relationships). In case the values assigned to the levels and fact relationships make the dimension model non-summarizable, the refactoring algorithm is the proposed resolution. We intend to study how to rewrite multidimensional conceptual queries in terms of the compiled views generated by our algorithm. Deciding which of the compiled views to materialize, if any, is a related challenge we need to address.

6.3.2 Extending Mapping Compilation Algorithm

In the current version of mapping compilation algorithm, we consider the minimum number of joins between tables as the optimization criterion in generating view queries. The search for finding such minimal set of tables that provides us with appropriate data is performed at schema level. However, the number of join operations that are done on

tables data as well as the available indices are also an important factor in efficiency of the generated views. Therefore, the size of tables, in terms of the number of tuples, should be taken into account. A larger number of tables joined together to make a join set can be a better option in generating views when the number of join operations among tables tuples is fewer than those of a smaller join set. In order to reflect this point on mapping compilation, we need to extend the definition of *minimalcoveringassociaion* such that a covering association is minimal considering both the number of tables in its join set and the number of join operations. As a result, the algorithm of finding minimal covering association should be modified to address this new definition of minimality. We can specify a weighted schema graph in which the labels connecting two tables is labeled by a number reflecting the number of joins between adjacent tables. Accordingly, the *Dijkstra* algorithm is performed on a weighted graph. This way, we consider both the number of tables and the number of join operations.

We also intend to support more cases of conditions on user mappings between conceptual attributes and data warehouse tables. A challenge could be taking into account the conditions in which an attribute is mapped to a function (MIN, MAX, AVG, ...) which is applied to a column. In this case, based on the complexity of the function the same algorithm or a modified version can support the correct view generation process.

Bibliography

[1] Mondrian project. `http://mondrian.pentaho.org/`.

[2] Atul Adya, José A. Blakeley, Sergey Melnik, and S. Muralidhar. Anatomy of the ado.net entity framework. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 877–888, New York, NY, USA, 2007. ACM.

[3] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 430–440, New York, NY, USA, 1987. ACM.

[4] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.

[5] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[6] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou, and Hyoung-Joo Kim. Data model issues for object-oriented applications. *ACM Trans. Inf. Syst.*, 5(1):3–26, 1987.

[7] D. S. Batory. Genesis: a project to develop an extensible database management system. In *OODS '86: Proceedings on the 1986 international workshop on Object-oriented database systems*, pages 207–208, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[8] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984.

[9] Leopoldo Bertossi, Loreto Bravo, and Monica Caniupan. Consistent query answering in data warehouses. *The Alberto Mendelzon Workshop on Foundations of Data Management (AMW2009)*, 450, 2009.

[10] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347, New York, NY, USA, 2006. ACM.

[11] Michael J. Carey, Donald D. Chamberlin, Srinivasa Narayanan, Bennet Vance, Doug Doole, Serge Rielau, Richard Swagerman, and Nelson Mendonça Mattos. O-o, what have they done to db2? In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 542–553, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[12] Michael J. Carey and David J. DeWitt. Of objects and databases: A decade of turmoil. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 3–14, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[13] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and file management in the exodus extensible database system. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 91–100, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[14] P.P. Chen. The entity-relationship model - toward a unified view of data. *ACM TODS*, 1(1):9–36, 1976.

[15] T. Connolly and C. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Addison Wesley, Essex, UK, 2009.

[16] George Copeland and David Maier. Making smalltalk a database system. *SIGMOD Rec.*, 14(2):316–325, 1984.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[18] Umeshwar Dayal and Philip A. Bernstein. On the updatability of relational views. In *VLDB'1978: Proceedings of the fourth international conference on Very Large Data Bases*, pages 368–377. VLDB Endowment, 1978.

[19] U. Deppisch, H.-B. Paul, and H.-J. Schek. A storage system for complex objects. In *OODS '86: Proceedings on the 1986 international workshop on Object-oriented database systems*, pages 183–195, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[20] Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient algorithm for computing bisimulation equivalence. *Theor. Comput. Sci.*, 311(1-3):221–256, 2004.

[21] Benjamin Eze, Craig Kuziemsky, Liam Peyton, Grant Middleton, and Alain Moutham. Policy-based data integration for e-health monitoring processes in a b2b

environment: experiences from canada. *J. Theor. Appl. Electron. Commer. Res.*, 5(1):56–70, 2010.

[22] Ronald Fagin, Laura M. Haas, Mauricio Hernández, Renée J. Miller, Lucian Popa, and Yannis Velegrakis. Clio: Schema mapping creation and data exchange. pages 198–236, 2009.

[23] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

[24] K. Gao and I. Pandis. Implementation of TPC-H and TPC-C toolkits. *At address: ¡www.cs.cmu.edu/ ipandis/courses/15823/project_final_paper.pdf¿.*

[25] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[26] S. Grimes. Object/relational reality check. *Database Programming & Design (DBPD)*, 11(7), 1998.

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

[28] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. *SIGMOD*, 25(2):205–216, 1996.

[29] Cindi Howson. *BusinessObjects XI (Release 2): The Complete Reference*. McGraw-Hill, Inc., New York, NY, USA, 2006.

[30] C. Hurtado and C. Gutierrez. Computing cube view dependences in olap datacubes. *SSDBM*, pages 33–42, 2003.

[31] C. A. Hurtado and Claudio Gutierrez. Capturing summarizability with integrity constraints in olap. *ACM Transactions on Database Systems*, 30(3):854–886, 2005.

[32] Carlos A. Hurtado, Claudio Gutiérrez, and Alberto O. Mendelzon. Capturing summarizability with integrity constraints in olap. *ACM Trans. Database Syst.*, 30(3):854–886, 2005.

[33] Carlos A. Hurtado, Claudio Gutiérrez, and Alberto O. Mendelzon. Capturing summarizability with integrity constraints in olap. *ACM Trans. Database Syst.*, 30(3):854–886, 2005.

[34] Carlos A. Hurtado and Alberto O. Mendelzon. Olap dimension constraints. *ACM)*, pages 169–179, June 2002.

[35] H. V. Jagadish, Laks V. S. Lakshmanan, and D. Srivastava. What can hierarchies do for data warehouses? *VLDB*, pages 530–541, 1999.

[36] D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. 1982.

[37] Anand S. Kambel. A conceptual model for multidimensional data. *APCCM*, 79, 2008.

[38] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *In ICDE*, pages 129–140, 2002.

[39] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *In ICDE*, pages 129–140, 2002.

[40] Ralph Kimball. *The Data Warehouse Toolkit*. John Wiley and Sons Inc., Indianapolis, Indiana, 1996.

[41] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75, 2005.

[42] Vishu Krishnamurthy, Sandeepan Banerjee, and Anil Nori. Bringing object-relational technology to the mainstream. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 513–514, New York, NY, USA, 1999. ACM.

[43] H. Lenz and A. Shoshani. Summarizability in olap and statistical data bases. *In Proceedings of SSDBM*, pages 132–143, 1997.

[44] H. Lenz and A. Shoshani. A survey on summarizability issues in multidimensional modeling. *Data and Knowledge Engineering Journal*, (68):1452–1469, 2009.

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

[46] M. Levene and G. Loizou. Why is the snowflake schema a good data warehouse design? *Information Systems*, pages 225 – 240, 2003.

[47] Jeff Linwood and Dave Minter. *Beginning Hibernate, Second Edition*. Apress, Berkely, CA, USA, 2010.

[48] Barbara Liskov. A history of clu. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 133–147, New York, NY, USA, 1993. ACM.

[49] Guy M. Lohman, Bruce Lindsay, Hamid Pirahesh, and K. Bernhard Schiefer. Extensions to starburst: objects, types, functions, and rules. *Commun. ACM*, 34(10):94–109, 1991.

[50] H. P. Luhn. A business intelligence system. *IBM J. Res. Dev.*, 2(4):314–319, 1958.

[51] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, 1979.

[52] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented dbms. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 472–482, New York, NY, USA, 1986. ACM.

[53] E. Malinowski and E. Zimnyi. *Advanced Data Warehouse Design: From Coventional to Spatial and Temporal Applications*. Springer, Berlin, 2008.

[54] Elzbieta Malinowski and Esteban Zimanyi. *Advanced Data Warehouse Design: From Conventional to Spatial and Temporal Applications*. Springer, 2009.

[55] J. N. Mazon, J. Lechtenborger, and J. Trujillo. Solving summarizability problems in fact-dimension relationships for multidimensional models. *DOLAP*, pages 57–64, 2008.

[56] Jose-Norberto Mazón, Jens Lechtenbörger, and Juan Trujillo. A survey on summarizability issues in multidimensional modeling. *Data Knowl. Eng.*, 68(12):1452–1469, 2009.

[57] S. Melnik, A. Adya, and P. Bernestein. Compiling mappings to bridge applications and databases. *Transactions on Database Systems*, 33(4), 2008.

[58] Sergey Melnik. Mapping-driven data access.

[59] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.*, 33(4):1–50, 2008.

[60] Fatemeh Nargesian, Flavio Rizzolo, Iluju Kiringa, and Rachel Pottinger. Refactoring dimension schemas for OLAP queries. *Technical Report*, 2010.

[61] James Ong, Dennis Fogg, and Michael Stonebraker. Implementation of data abstraction in the relational database system ingres. *SIGMOD Rec.*, 14(1):1–14, 1983.

[62] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[63] T. B. Pedersen, C. S. Jensen, and C. E. Dyreson. Extending practical pre-aggregation in on-line analytical processing. *VLDB*, pages 663–674, 1999.

[64] Torben Bach Pedersen, Christian S. Jensen, and Curtis E. Dyreson. A foundation for capturing and querying complex multidimensional data. *Inf. Syst.*, 26(5):383–423, 2001.

[65] D. J. Power. A brief history of decision support systems. In *DSS Resources, World Wide Web*, page 2003, 2003.

[66] M. Rafanelli and A. Shoshani. Storm: A statistical object representation model. *In Proceedings of SSDBM*, pages 14–29, 1990.

[67] Flavio Rizzolo, Iluju Kiringa, Rachel Pottinger, and Kwok Wong. The conceptual integration modeling framework: Abstracting from the multidimensional model. *Technical Report*, 2010.

[68] J. Lechtnborger S. Rizzi, A. Abello and J. Trujillo. Research in data warehouse modeling and design: dead or alive? *Proceedings of the 9th ACM international workshop on Data warehousing and OLAP*, (68):3–10, 2006.

[69] Carsten Sapia, Markus Blaschka, Gabriele Höfling, and Barbara Dinter. Extending the e/r model for the multidimensional paradigm. In *ER '98: Proceedings of the Workshops on Data Warehousing and Data Mining*, pages 105–116, London, UK, 1999. Springer-Verlag.

[70] George Spofford, Sivakumar Harinath, Christopher Webb, and Francesco Civardi. *MDX Solutions: with Microsoft SQL Server Analysis Services 2005 and Hyperion Essbase*. John Wiley & Sons, Inc., New York, NY, USA, 2005.

[71] Michael Stonebraker. Object management in postgres using procedures. In *OODS '86: Proceedings on the 1986 international workshop on Object-oriented database systems*, pages 66–72, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[72] CORPORATE The Paradise Team. Paradise: a database system for gis applications. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, page 485, New York, NY, USA, 1995. ACM.

[73] N. Tryfona, F. Busborg, and J. G. B. Christiansen. starer: A conceptual model for data warehouse design. *In Proceeding of ACM 2nd Int. Workshop on Data Warehousing and OLAP (DOLAP)*, 1999.

[74] Dan Volitich. *IBM Cognos 8 Business Intelligence: The Official Guide*. McGraw-Hill, Inc., New York, NY, USA, 2008.