Supporting Conceptual and Storage Multidimensional Integration for Data Warehouse Creation

Fatemeh Nargesian[a], Flavio Rizzolo[a], Iluju Kiringa[a], Rachel Pottinger[b]

[a] *University of Ottawa*
[b] *University of British Columbia*

Abstract

This paper introduces the Storage and Conceptual Integrated Multidimensional Model (SCIMM) Framework, a requirements-driven data warehouse construction approach. The SCIMM Framework allows users to raise the level of data integration abstraction by specifying their needs at a high conceptual level which is then implemented in a multidimensional platform.

Relationships between the conceptual model and multidimensional data are specified in the SCIMM by a set of attribute-to-attribute correspondences. These are compiled into a set of mappings that associate each construct in the conceptual model with a query on the physical model.

This paper focuses on three steps of our vision: (1) the overall SCIMM framework for allowing the user to create a conceptual model, (2) how to compile the given correspondences into queries over the physical model, and (3) how to refactor the conceptual model into summarizable dimensions.

Keywords: Conceptual Modeling, Data Warehouse and Repository

1. Introduction

Almost every aspect of today's world, from business, to science, to digital social interactions, is drowning under a deluge of data. The amount of data in the world grows at an astounding 60% [1] annual rate. The only way to cope with this deluge is to move toward a data-centered way of conducting human activities — a way that focuses on making sense of the data.

For example, Walmart has over 200 million transactions weekly [1]. On the one hand, gaining insight into the retailer's business requires using enormous multidimensional data warehouses, i.e., largely static repositories of data gathered for analysis. Data warehouses commonly use a multidimensional view of data, which allows the basic data (stored in fact tables) to be analyzed according to various hierarchies (e.g., location, time). One key aspect of this problem is to look at data warehouse construction. A typical data warehouse construction process is something like the following:

- A business executive asks questions about the data which is stored in a transactional database.
- The IT professional attempts to answer the questions over the database. These questions turn out to be complex and slow to be answered over the existing database.
- The IT professional spends weeks or months creating a data warehouse to analyze the data to answer all the kinds of questions that the business executives want to have answered. This warehouse is huge but comprehensive.

Email addresses: fnarg012@site.uottawa.ca (Fatemeh Nargesian), frizzolo@site.uottawa.ca (Flavio Rizzolo), kiringa@site.uottawa.ca (Iluju Kiringa), rap@cs.ubc.ca (Rachel Pottinger)

- The business executive uses some of the warehouse results, but any changes from these details have to be run through the IT professional.

We advocate a different model, which encourages a more user-centric approach as well as only materializing the parts of the data warehouse that are used — The first bullet is the same, but the rest are different:

- A business executive asks questions about the data which is stored in a transactional database.
- The business executive spends some time creating a desired specification in a business-level language.
- This business-level language specification is transformed into a conceptual model (like ER diagrams, only for multi-dimensional data).
- The conceptual model is mapped to a storage model (like the relational model).
- The storage model is mapped to the transactional data sources.
- At query time, the business executive queries over the business-level specification, and the data may either be freshly gathered from the data sources or materialized into a data warehouse.

Our solution has multiple strengths, including (1) the business executive has an easier time working closely with the data, since the business executive can specify what is desired, not the IT professional (2) it is faster to deploy the solution; because only a small amount of the system will be materialized, the design does not have to be as carefully optimized — mistakes will be smaller, and the system will be more flexible and capable of being changed and (3) the amount of space needed on disk will be reduced dramatically.

Materializing more of a data warehouse than is needed is a major problem in business intelligence/business analytics. Developers at SAP, a leading business analytics firm, estimate that only 10% of the data that they create in data warehouses is eventually consumed — this is a huge issue and causes great problems to those trying to perform business analytics.

This is a huge task; there are many sub-problems that must be tackled. Figure 1 shows the full SCIMM Framework: a business level which is converted to a conceptual level, which is converted to a storage level which then gathers the data as needed from the sources..

In this paper we present the first step on the road to this vision: how to create a data warehouse conceptual model that can be related to a storage model in what we call the *Storage and Conceptual Integrated Multidimensional Model (SCIMM) Framework*. In this initial step, we assume that there exists a back-end data store as is shown in Figure 2. However, even with this assumption, there are substantial research challenges to tackle, including determining the various representations and how to translate between them, as we detail throughout this paper. We motivate the three main challenges tackled in this paper: defining the system framework, compiling mappings from the conceptual to storage level, and refactoring data in Sections 1.1, 1.2, and 1.3, respectively.

1.1. SCIMM Framework

Our SCIMM model is based on the EDM Framework [2]. Both provide conceptual levels, storage levels, and mappings between the two. However, whereas EDM provides structure for the relational database model, our SCIMM provides structure for the relational multi-dimensional model. Each of the different levels has two different forms: a *visual* language, which allows the user to easily see the concepts involved and a *data* language, which contains all of the details necessary to translate query and data. We explain each of the models in detail in Section 3.

Example 1.1. *Dream Home [3] is a fictitious property management firm that brokers deals between potential tenants/buyers and owners wishing to rent/sell their properties. The organization has branches throughout North America.*

Figure 3 shows a small fragment of the Dream Home data warehouse expressed in the Storage and Conceptual Integrated Multidimensional Model (SCIMM) [4]. The SCIMM model of Figure 3 offers both a conceptual model of the data, on the left-hand side, and a storage model of the data, shown on the right-hand side.

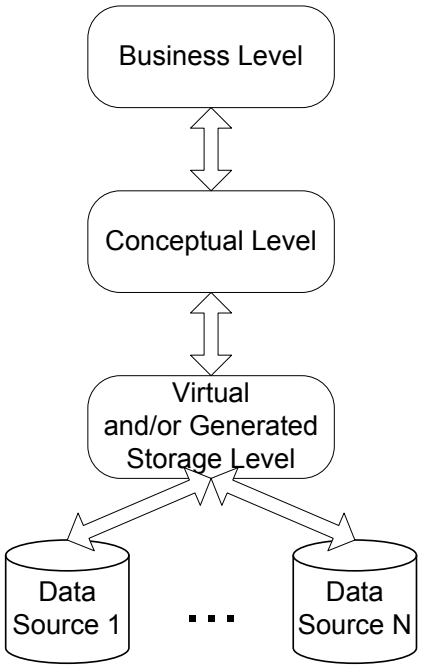


Figure 1: The planned overall SCIMM Framework. The user communicates at a business-level by asking business queries. This is turned into a conceptual layer (like ER, but for multidimensional models). This is then transformed into the storage level, which may either be materialized or virtually constructed from the individual sources.

In the conceptual level in Figure 3, Canadian Location *and* Client *(shadowed rectangles) are dimensions describing measures in the* Sale *fact table (shadowed diamond). Non-shadowed rectangles (e.g.,* Branch*,* City*, and* Client*) represent levels in the dimensions. These levels are related by parent-child relationships, which are drawn as arrows from more specific levels to more general levels. Some directly related concepts (e.g.,* Area *and* Region*) are also directly related in the respective storage tables, whereas others are indirectly related in the storage model via third tables (e.g.,* Region *and* Country*). Annotations on the dashed arrows are basically filters on attribute values of the storage model.*

1.2. Mapping Compilation

A fundamental role of SCIMM is to make the conceptual multidimensional model an executable component that can be used at run-time. EDM [2] similarly takes a conceptual model for a relational database and changes it into a storage model. We build upon the EDM example to perform the same service for multi-dimensional data warehouses: given a set of correspondences (such as those in Figure 3) we create executable views which allow data from the store level to be returned in response to queries from the conceptual level.

Defining those views over the storage level is not an easy task: it requires a deep understanding of the conceptual model and the physical organization of the data warehouse. Hence, it is not practical for a high-level user accustomed to only the conceptual view of the data to come up with such complex view definitions. Our solution requires the user to just provide very simple *correspondences* between attributes in both models (with or without value conditions), which are later transparently compiled by the system into fully-fledged, multidimensional mappings that can be used for query evaluation.

For instance, the storage level SCity may contain every city in the database, even those without branches. However, the analyst of the example wants to have in the conceptual level City only those cities that make sense in the Canadian Location dimension, i.e., those with branches. By using SCIMM instead, the analyst draws just two correspondences from cityID and name, and the tool takes care of creating the view, as we show in greater detail in Section 5.

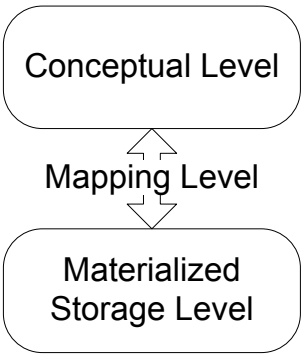


Figure 2: The current SCIMM Framework focused on in this paper. As a first step, we assume that there exists a materialized data warehouse. In this paper, we concentrate on how to transform from the conceptual to the storage level and vice versa.

1.3. Refactoring

A key strategy for speeding up query answering in a multidimensional environment is to reuse cube views with precomputed aggregations. In order to compute the aggregated cube for a level, we need to consider every instance of the bottom categories in the aggregation process exactly once. This is only possible when the dimensions used in the cube view definitions are *summarizable*. Summarizability is a desirable condition in most OLAP applications because it allows for aggregations at finer levels in a dimension to be used to compute aggregations at higher levels. Even though column stores and in-memory data warehouses are becoming more common, most data warehouses rely on cube views for speeding up query answering.

Example 1.2. *Consider a USA-Canada location dimension for the Dream Home running example in which every city rolls up to either a* Province *(in Canada) or a* State *(in the US), but not to both. Some cities may even roll up directly to* Country *(e.g., Washington DC). As a result, no single level cube view can be used for query evaluation: the cube for* Province *is missing all cities in the US and the one for* State *is not aggregating any city in Canada. Consequently, such a dimension is non-summarizable.*

A non-summarizable dimension can be converted to one or more summarizable ones by fixing their instances [5]. In contrast, in this paper we propose a methodology to transform a non-summarizable dimension into a summarizable one by changing the schema. We call this methodology *refactoring* and describe it in depth in Section 6.

1.4. Contributions and Outline

- We describe the Storage and Conceptual Integrated Multidimensional Model (SCIMM) Framework, which allows top down data warehouse creation; while the visual languages are not a major contribution, using such a full suite of languages and what the suite of languages allows, is;
- We define the SCIMM framework and describe how it can handle all the necessary transformations;
- We provide an algorithm to create mappings between conceptual and logical multidimensional models. The algorithm compiles simple user-provided attribute-to-attribute correspondences into complex views over the multidimensional model. We experimentally validate the viability of the compilation algorithm on synthetic and real-world data.
- We provide an algorithm for refactoring dimensions.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 gives an overview of the different levels of the SCIMM framework. Section 4 formally describes the conceptual model and the mappings' semantics and syntax. Section 5 presents the mapping compilation language and semantics and the compilation algorithm. Section 6 describes the refactoring process. Section 7 presents our implementation and experimental results. Section 8 concludes.

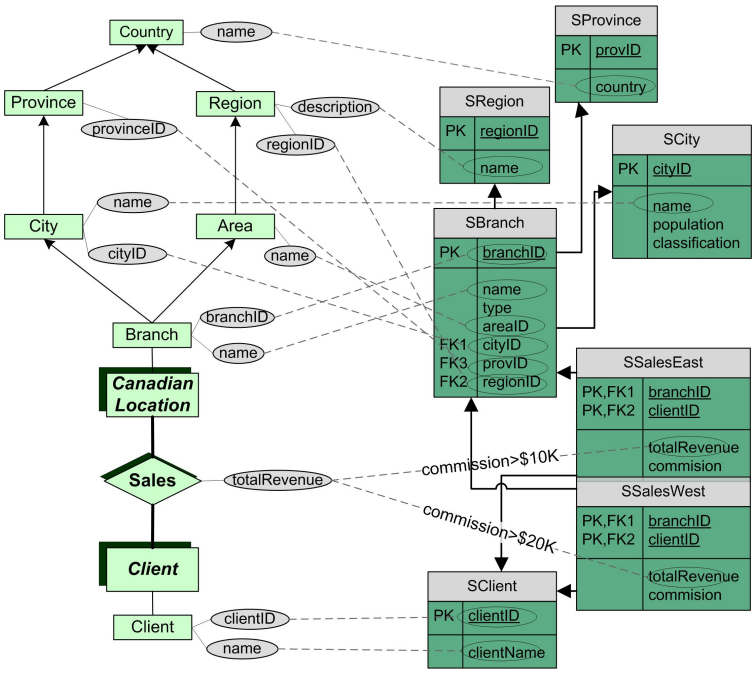


Figure 3: Dream Home SCIMM Visual Model: Conceptual Level (left), Store Level (right) and Mapping Level (dashed lines)

2. Related Work

Kimball *et al.* [6] note that the bottom-up warehouse construction contrasts with a top-down strategy which the authors doubt could be possible; they go as far as to assert that using a conceptual model to that end is infeasible. Contrary to the assertion in [6], the last ten years have witnessed some data warehouse design approaches which include a conceptual design phase [7, 8].

There are several requirements-driven approaches for data warehouse creation. GRAnD [9] focuses on the design of conceptual model from requirements analysis. Our SCIMM framework focuses on facilitating data warehouse construction by having the conceptual model drive the underlying storage level. Therefore, the two projects aim to solve different sub problems of a same super problem — building a requirement-driven data warehouse. However, the conceptual models for the two projects are different. GRAnD places "interesting" attributes, which are in the data source but not specifically required by the users, in the conceptual model and labeled as "supplied", while we only put user requested attributes in the conceptual model. The GRAnD approach will result in materializing more of a data warehouse than needed. However, a possible risk for our approach is that user may miss some needed attributes in the requirement analysis phase. Our solution is to provide the SL_V which contains all the information about source data, and a simple modification of the CL_V and the mapping to the SL_V can add the missing attributes in the data warehouse.

Mazón et al. [10, 11] present a hybrid approach to modeling a data warehouse. The framework uses both the goals of decision makers and the data sources as input. The goals are categorized into information level requirements then automatically mapped to relevant structures in the data sources to build an initial estimation of the data warehouses conceptual model. The conceptual model is further altered to better reflect the data sources and validated using the three Multidimensional Normal Forms [10]. Similarly, Romero and Abelló have developed a hybrid approach to multidimensional conceptual model development. Their work in [12] guides a designer to formalize user requirements as SQL queries, so the multidimensional knowledge present in the queries and data sources may be used to automatically generate a conceptual model. In [13], they present a step to assist with the requirements elicitation stage by performing an ontology-based evaluation of the data sources and presenting potential interesting subjects of analysis to the user.

In contrast, the conceptual model of the SCIMM Framework allows the user to represent the data warehouse in a way that is intuitive to them, rather than constraining the conceptual model to represent the existing data sources. With this additional flexibility, our solution allows for a more dynamic environment where users may alter the conceptual

model on demand.

Hahn *et al.* automatically generate OLAP schemas from conceptual graphical models [14] using multidimensional extension of ER [15]. Malinowski and Zimànyi describe a multidimensional model called MultiDim for representing OLAP and data warehouse requirements [8], which is similar to ER. They emphasize capturing the various sorts of (often irregular) hierarchies that appear in real-world applications. Malinowski and Zimànyi also provide mapping rules for implementing MultiDim schemas into the object-relational logical model. This mapping is based on the classical rules for mapping ER models to the relational model.

The above graphical approaches are mainly proposals for modeling languages made as part of data warehouse design methodologies. They are rarely used as a run time environment. By contrast, our proposal provides a run-time environment that allows a user to pose queries at a conceptual level that is abstracted from the logical multidimensional data level.

As mentioned in Section 1, EDM [16, 2] raises the level of abstraction from the logical relational data level to ER's conceptual level. EDM's goal is similar to ours; however, they work on relational models, and our goal is to work on multidimensional ones. Hibernate [17] is a framework to make Java classes persist. At design time, the developer writes (in Java) classes and mappings of those classes to SQL queries over an underlying relational schema. These mappings are not compiled; they solely translate objects to SQL queries.

Like our SCIMM Framework, EDM and Hibernate aim to bridge the gap between the object-oriented world of application programmers and the world of relational data. Unlike our SCIMM Framework, which deals with multidimensional data, EDM and Hibernate deal with classical relational data.

In industry, SAP Business Objects and IBM have proprietary conceptual levels. SAP Business Objects' Universes [18] represent an organization's data assets (i.e., data warehouse as well as transactional databases). The universe hides the complex structure of the underlying data assets as well as their provenance and localization from the end user by providing the later with a familiar business-oriented interface. IBM's Framework Manager [19], is similar to SAP's Universes and works according to the same principles.

3. Storage and Conceptual Integrated Multidimensional Model (SCIMM) Framework

3.1. SCIMM Overview

As shown in Figure 2, this paper focuses on the Conceptual Level (CL) and Storage Levels (SL) of the SCIMM framework, as well as the Mapping Level (ML) in-between. Each of these levels has two different aspects: the *visual* aspect and the *data* aspect. The visual aspect is the part that the users interact with; e.g., in Figure 3 shows the visual aspects of the CL, ML, and SL, which we refer to as CL_V, ML_V and SL_V respectively. Since more detailed information is required in order for the system to be operational, data aspects, for each of the conceptual, mapping, and storage levels, which we refer to as CL_D, ML_D, and SL_D respectively are stored in XML and enable interoperability with business intelligence applications.

This separation of the CL, ML, and SL layers parallels similar layers in the EDM Framework [2]. However, whereas EDM creates structure for relational data, and our SCIMM creates structure for multi-dimensional data. Thus, a user can design concepts in CL_V that are automatically translated into CL_D for run-time usage.

In Sections 3.2, 3.3 and 3.4 we briefly describe the conceptual, storage, and mapping levels respectively. Section 3.5 discusses the overall framework.

3.2. The Conceptual Level

Our CL_V model extends MultiDim [8], which in turn extends Peter Chen's ER model [15] to support multi-dimensionality. This section briefly reviews conceptual data modeling of multi-dimensionality and introduces CL_V. Readers interested in more details of adding multi-dimensionality to conceptual data modeling should refer to another source, e.g., [8].

Typically, data warehouses are described and defined by a multidimensional model in which data are points called *facts* in an n-dimensional *data cube*. The *dimensions* of a data cube are the perspectives used to analyze data and consist of a set of *levels of granularity*. Facts are represented by *fact relationships*, which relate entities in different dimensions.

We make these features more concrete by looking at the example given in Figure 3. The CL_V Canadian Location and Client (shadowed rectangles) are dimensions describing measures in the Sale fact table (shadowed diamond). Non-shadowed rectangles (e.g., Branch, City, and Client) represent levels in the dimensions. Unlike MultiDim, dimensions are first-class citizens in CL_V diagrams, and thus are explicitly represented. Levels and fact relationships may also have properties/measures, represented by ovals (e.g., branchID, name). The directly attached level to a dimension is the bottom level of that specific dimension, which determines the granularity of the facts in the respective fact relationship. For instance, the totalRevenue values in Sales are given by Branch and Client.

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 Canadian Location indicates that sales can be analyzed either by geographical location (starting at Branch and following the City, Province and Country path) or real estate market (starting also at Branch and following the Area, Region and Country path). These are two complementary ways of aggregating sales data, since areas and regions may span parts of two cities, and even two provinces, e.g., the National Capital Region spans large areas of Ottawa (Ontario) and Gatineau (Quebec) but they do not cover either city entirely. This figure is designed only to show the location hierarchy for Canada, e.g., the Country entity contains only one instance with name="Canada".

The CL_D largely mirrors the CL_V, only in an XML format for easy manipulation. For example, Figure 4 shows the CL_D representation of the Branch level from Figure 3.

```
- <level name="Branch">
    <property name="branchID" type="String" />
    <property name="name" type="String" />
  </level>
```

Figure 4: CL_D Branch Level

3.3. The Storage Level

The SL_V is a UML-like representation of the relational data warehouse schema, containing relational table definitions, keys, and referential integrity constraints. As mentioned in the introduction, in our current practice, the SL specification is automatically generated from the underlying data warehouse schema. Since the eventual goal is to not have an existing data warehouse, at that point the SL will be generated through another technique.

As with the conceptual level, the SL_D largely mirrors the SL_V, only in an XML format for easy manipulation. For example, Figure 5 shows the SL_D representation of the Branch dimension table from Figure 3.

```
- <dimension name="SBranch">
    <key name="branchID" />
    <column name="branchID" type="int" />
    <column name="name" type="nvarchar" />
    <column name="type" type="nvarchar" />
    <column name="areaID" type="int" />
    <column name="cityID" type="int" />
    <column name="provID" type="int" />
    <column name="regionID" type="int" table="SRegion"
      column="regionID" />
    <foreign-key name="cityID" table="SCity" column="cityID" />
    <foreign-key name="provID" table="SProvince" column="provID" />
    <foreign-key name="regionID" table="SRegion" column="regionID" />
  </dimension>
```

Figure 5: SL_D Branch dimension table

3.4. The Mapping Level

There is a conceptual gap between the CL_V and SL_V representations. Sales are organized into two fact tables in the SL_V (SSalesEast and SSalesWest) whereas the conceptual model has a single fact relationship called Sales. There are other differences as well: some roll up functions are represented implicitly by two columns in the same table (e.g., the

roll up between branch and area is given by branchID and areaID columns in the SBranch table) and others are specified by foreign keys (e.g., the roll up between branch and city is given by the cityID foreign key in SBranch). In contrast, the CL_V represents all roll up functions explicitly as parent-child relationships between levels.

As previously mentioned, the user-provided correspondences between the CL_V and SL_V (i.e., the ML_V) can include value conditions specifying the attribute values for which the correspondence holds. For example, the correspondences from totalRevenue in Sales specify the commission values for which the total revenue will be included in the Sales fact relationship.

From the ML_V, the SCIMM mapping compilation creates views over the SL_D for every level, parent-child relationship and fact relationship in the CL_V and adds the SQL view definitions to the ML_D. ML_D mappings integrate conceptual elements that potentially distributed in more than one store table. The views that we create are similar to those in data integration [20] or data exchange [21]. In particular, SCIMM uses mappings of the common form $c \rightsquigarrow \psi_s$, where, as in data exchange, c is an element in the *target* schema and ψ_s is a query (or view) over the *source* schema. Intuitively, the expression $c \rightsquigarrow \psi_s$ means that the schema element c is associated with the view ψ_s. These mappings are part of the ML_D. In SCIMM, the conceptual data model (CL_D) functions as the target schema and the store data model (SL_D) as the source schema. The element c in the mapping expression is either a level, a parent-child relationship, or a fact relationship in the CL_D whereas ψ_s is a view over the data warehouse's multidimensional tables in the SL_D. Note that both the CL_D element c and its SL_D view ψ_s need to have the same schema.

The semantics of $c \rightsquigarrow \psi_s$ is given by the correspondences, their value conditions and all applicable SL_V integrity constraints. If c has correspondences to a single table r, the definition of the mapping is straight-forward: ψ_s is defined over r and its tuples are those in r satisfying the correspondence value conditions. If c has correspondences to tables $r_1, \ldots, r_n$, then ψ_s is defined over $r_1, \ldots, r_n$ and its tuples are those not only satisfying the correspondence value conditions but also the integrity constraints among the tables.

Example 3.1. *Consider an ML_D mapping m :* Region $\rightsquigarrow \psi_s$. *View ψ_s selects* regionID *column from* SBranch *and* name *column from* SRegion. *Since there is a foreign key between the two tables, tuples are selected from the join result of* SBranch *and* SRegion. *The SQL expression for view ψ_s is the following:*

```
SELECT SBranch.regionID AS regionID, SRegion.name AS name
FROM SBranch, SRegion
WHERE SBranch.regionID = SRegion.regionID
```

In our current implementation the ML_V is created by users creating correspondences between elements in the CL_V and SL_V. However, this is not the focus of this paper and we refer the reader to other literature on creating correspondences between schema elements (e.g., the survey at [22]).

3.5. SCIMM Framework Discussion

Conceptual and physical/storage representations tend to evolve differently. Business analysts may want to change the conceptual representation on the fly without having to request costly changes to the physical organization of the data. For instance, they may later decide that having two separate fact relationships for East and West Sales better suits their analysis needs and decide to change the CL_V of the figure accordingly. Furthermore, different analysts may need different conceptual views of the same data warehouse, thus there may simultaneously need to be multiple conceptual models for the same physical storage.

Traditional methodologies use the conceptual model almost exclusively for data warehouse design, never as an implementable data model against which queries are posed. In contrast, the SCIMM framework offers both design time and run time environments that implement a conceptual modeling language. In addition, SCIMM users can specify relationships between the conceptual model and multidimensional data by defining attribute-to-attribute correspondences. These are compiled into a set of mappings that associate each construct in the conceptual model with a query on the logical model that can be used at run time.

4. Formalizing Schemas and Mappings for View Compilation

4.1. SCIMM schemas

To represent the SL_V schema during view compilation, we create a *schema graph*:

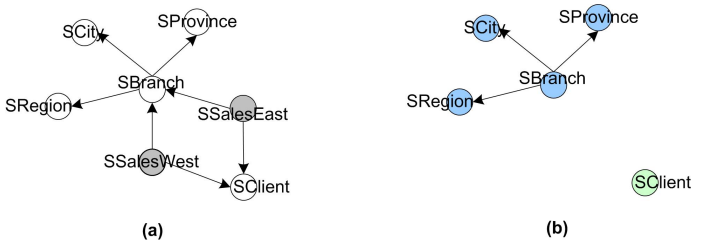


Figure 6: (a) Complete store schema graph for our running example; (b) Its dimension subgraph with two dimension lattices

Definition 4.1. *Given SL_V 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* **store 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 store schema graph is* **complete** *if $\forall$ tables t in S, $\exists$ a node n in N s.t. $\chi(t, n)$ and $\forall$ referential constraints $c \in S$ $\exists$ edge e in E s.t. $\chi(c, e)$.*

Within a store schema graph we distinguish a subgraph we call a *store dimension graph*:

Definition 4.2. *Let $G = \{N, E\}$ be the complete store schema graph of an SL_V schema S. We distinguish two set 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 $G_D = \{D, E_D\}$ the* **dimension subgraph** *of G. 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.*

This definition captures how SL_V tables are related via foreign key constraints. Note that every edge in a dimension subgraph corresponds to a roll up function, but there may be roll up functions that are not associated to edges (e.g., when parent and child levels are attributes in the same SL_V table).

The SL_V does not impose any conditions on how dimension 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 [23, 8]. Such 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 with partially normalized data (starflake schemas) are also common.

We formalize next the notion of a *well-formed SL_V* schema that captures all the patterns mentioned above.

Definition 4.3. *Let $G = \{N, E\}$ be the complete store schema graph of an SL_V schema S and G_D be its dimension subgraph. We say that SL_V 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) *the composition of roll up functions is* strict*, i.e., every instance of a level cannot reach more than one instance in a higher level in G;* (4) *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;* (5) *each dimension lattice has a unique bottom node;* (6) *for every edge $\langle n, d \rangle \in E$, $d \in D$; and* (7) *for every $\langle f, d \rangle \in E$, $f \in F$, $d \in D$, d is the bottom of some dimension lattice.*

Conditions (1), (2) and (3) guarantee summarizability [24, 5, 25]. A dimension is *summarizable* if aggregations at a hierarchy's finer levels can be used to compute aggregations at higher levels. That only happens when all bottom level instances are considered in the aggregations of each ancestor level exactly once. Conditions (4) and (5) state that dimensions (comprised by one or more dimension tables) are disjoint and have a unique top and bottom. If there is more than one path between two nodes in a hierarchy, summarizability plus condition (5) guarantees that aggregating following any such paths provides the same correct result. That helps us choose what path to use for compiling views spanning several tables.

Example 4.1. *Suppose that the* Canadian Location *dimension in the CL_V of Figure 3 is instead a dimension lattice of a well-formed SL_V schema. There are two possible paths to go from* Branch *to* Country*: one through* Area *and* Region *and the other through* City *and* Province*. Since the schema is summarizable, we know that all branches are going to be aggregated in every level above. In consequence, aggregating via either of the two paths provides the same answer.*

Finally, condition (6) forbids edges between two fact table nodes and condition (7) specifies that every fact table is connected to the bottom of some dimension lattice. Note that the SL_V in Figure 3 is well-formed.

4.2. Mapping Compilation Components

Mapping compilation takes as input a pair of CL_D and SL_D schemas and an ML_D defining correspondences between them. Then the compilation process generates a set of views that map CL_D schema elements to SL_D 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.

Our ML_D views are expressed in SQL. The following definitions describe the components needed to build such views. We start by characterizing the set of tables that appear in the view definition's **FROM** clause, which we call a *join set*:

Definition 4.4. *A* **join set** $J = \{j_1, \ldots, j_n\}$ *over SL_D 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. *It is easy to see that tables* SSalesWest, SSalesEast, 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 (arrows on the right side of Figure 3).*

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

A set of correspondences between two entities form an ML_D *mapping fragment*. E.g., there is a single ML_D mapping fragment between CL_D Branch and SL_V SBranch consisting of two correspondences, one for each CL_D Branch attribute.

The compiled SQL view joins the tables in a join set using the foreign keys and value conditions specified in the correspondences. Together, a join set and its join constraints define an *association*:

Definition 4.6. *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 corresponding constraint in C.*

Example 4.3. *Consider the join set J=*{SSalesWest, SSalesEast, SBranch, SRegion} *and set of constraints C =*{SSalesWest.branchID = SBranch.branchID, SSalesEast.branchID = SBranch.branchID, SBranch.regionID = SRegion.regionID, SSalesWest.commission > $20K, SSalesEast.commission > $10K}*. The SQL query fragment for association $\langle J, C\rangle$ is:*

```
FROM SSalesWest, SSalesEast, SBranch, SRegion
WHERE SSalesWest.branchID = SBranch.branchID
  AND SSalesEast.branchID = SBranch.branchID
  AND SBranch.regionID = SRegion.regionID
  AND SSalesWest.commission > $20K
  AND SSalesEast.commission > $10K
```

The **FROM** *clause contains the members of join set J and the* **WHERE** *clause is a conjunction of the constraints in C.*

Definition 4.7. *An association $\langle J, C\rangle$ is* **covering** *with respect to a set M of ML_D 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 SL_D table of m_i is in J, (2) all conditions in m_i appear as constraints in C.*

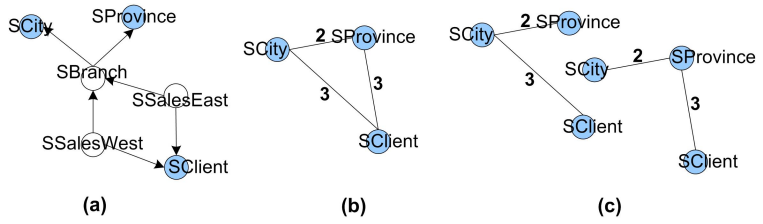


Figure 7: (a) Join graph for a covering association; (b) Compressed graph; (c) Its two minimal spanning trees

Example 4.4. *Consider the correspondences linking attributes* Region.regionID *and* Region.description *to columns* SBranch.regionID *and* SRegion.name*, respectively. These correspondences result in one ML_D mapping fragment between* Region *and* SBranch*, and another between* Region *and* SRegion*, without conditions. The association $\langle J, C\rangle$ in Example 4.3 is covering w.r.t. these two mapping fragments because tables* SBranch *and* SRegion *belong to the join set J.*

Definition 4.8. *A covering association $\langle J, C\rangle$ w.r.t. a set of ML_V mapping fragments M is* **minimal** *iff it is impossible to define a covering association $\langle J', C\rangle$ w.r.t. M where J' has smaller cardinality than J.*

Example 4.5. *The covering association $\langle J, C\rangle$ in Example 4.3 is not minimal. The minimal covering association w.r.t. the mapping fragments in Example 4.4 is $\langle J', C'\rangle$, where $J'=\{$*SBranch*,* SRegion *$\}$ and $C'=\{$*SBranch.region = SRegion.regionID *$\}$. The SQL fragment for $\langle J', C'\rangle$ 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 in the compiled views. Compiling an ML_D mapping m from a given CL_D entity c to an SL_D S requires two steps: (1) identify the ML_D mapping fragments in which c participates, (2) compute the minimal covering association w.r.t. to those mapping fragments. The result is a mapping $m: c \rightsquigarrow \psi_S$.

Example 4.6. *Consider an ML_D $m:$* Region *$\rightsquigarrow \psi_S$. The* **SELECT** *clause is given by attributes involved in the mapping fragments* Region-SBranch *and* Region-SRegion*. The* **FROM** *and* **WHERE** *clauses come from the minimal covering association in Example 4.5. The resulting SQL expression for view ψ_S is:*

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

5. Mapping compilation

As described in Section 4, creating an ML_D mapping is equivalent to finding a minimal covering association (Definition 4.8). To compute a minimal covering association w.r.t. to a set of ML_D mapping fragments $f_1, \ldots, f_n$ we first find the well-formed join set for the association (Definition 4.4), beginning by finding the set of SL_D 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 constraint. In that case, a covering association must include additional "connecting" tables, as shown in the following example.

Example 5.1. *Suppose that CL_D* Client *in Figure 3 has two additional attributes,* country *and* city*, that are mapped to* SProvince.country *and* SCity.cityID*, respectively.* Client *would participate in three ML_D mapping fragments:* Client-SClient*,* Client-SCity *and* Client-SProvince*. The SL_D tables in those mappings fragments (*SClient*,* SCity *and* SProvince*) 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 instance, adding*

```
COMPILE_MAPPINGS(S, C, M)
Input: SL_D schema S, CL_D schema C, and ML_D 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 8: Compile Mappings Algorithm

SBranch, SSalesEast *and* SSalesWest *to the join set makes it well-formed. Figure 7(a) shows the six nodes defining a well-formed join set. The three darker nodes correspond to SL_D tables in the mapping fragments; the other three 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.*

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.

Definition 5.1. *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 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 [26] to compute the shortest paths between pairs of nodes. The length of the shortest path is the weight of the edge connecting two nodes.

We use the well-known *minimum spanning tree* algorithm [26] to find a subgraph of the compressed graph that connects all the nodes so that the sum of the weights 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 weighs 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. We discuss the mapping compilation algorithm next.

Example 5.2. *Figure 7(c) shows the two possible minimum spanning trees of the compressed graph in Figure 7(b). 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.*

Finding paths in an SL_D schema graph resembles following a chase in a database [27]. Our mapping compilation algorithm finds a subset of chases that span a minimum set of store schema tables. In general, the chase may not

```
GET_MAPPING_FRAGMENTS(C, M, X)
Input: CL_D schema C, ML_D schema M, element X in C
Output: The set of mapping fragments for X
(1)  if X is a level;
(2)      MC ← search M for all map. frag. mf containing X;
(3)      return MC
(4)  if X is a parent-child relationship
(5)      MP ← search M for all map. frag. mf_p
             containing the parent level in X;
(6)      MC ← search M for all map. frag. mf_c
             containing the child level in X;
(7)  return MP ∪ MC;
(8)  if X is a fact relationship
(9)      MC ← search M for all map. frag. mf containing X;
(10)     foreach dimension D referenced in X
(11)         MD ← search M for all map. frag. mf
                 containing the bottom level of D
(12)         MC ← MC ∪ MD
(13) return MC;
```

Figure 9: Get Mapping Fragment Algorithm

terminate, and computing a chase can be exponential in the worst case [28]. However, our join paths are restricted by the well-formed structure of a data warehouse (Definition 4.4). This restriction guarantees the termination and polynomial time complexity of our algorithm, even in the worst case.

The mapping compilation algorithm (Figure 8) starts by creating a complete schema graph from the SL_D (line 2); this graph is used through the entire compilation process. Then it creates an ML_D view for each schema element in the CL_D model (lines 3-17) as follows. First, it identifies the mapping fragments relevant to the current CL_D schema element (line 4). Second, it finds the minimal covering associations for the mapping fragments (Section 4). This involves (i) creating the compressed graph (line 5), (ii) computing the minimum spanning tree (line 6), and (iii) expanding such a tree to obtain a schema graph whose nodes are the join set of the minimal covering association for the view (line 7). The set of constraints C of A is obtained by combining the join conditions in XT and the conditions from the mapping fragments MF (line 8). Finally, it translates the minimal covering associations to SQL view expressions, as shown in Examples 4.5 and 3.1, by invoking one of three algorithms, depending on the type of element: if the CL_D 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 mapping fragments gatherer (Figure 9) gets the relevant ML_D mapping fragments depending on the input CL_D element: (1) for levels (lines 1-3) it gathers the fragments containing the levels; (2) for parent-child relationships (lines 4-7) it gathers both the parent and child levels, since both are needed to create the view; (3) for fact relationships (lines 8-13) it finds the mapping fragments containing all the bottom levels of all dimensions linked to the fact relationship.

Functions COMPILE_LEVEL, COMPILE_PC_REL and COMPILE_FACT_REL create the SQL expression of the views by first compiling the subexpressions and then using UNION to combine them into a single expression. COMPILE_LEVEL (Figure 10) achieves this by including the column.table combinations in the **SELECT** part of the view (step (2)). The mapping fragment specifies these column.table combinations as the table and column to which a CL_D attribute is mapped. The keyword "AS" in step (2) specifies which column of an SL_D table is mapped. As discussed, users might also put selection restrictions on the mappings fragments. Thus, the join set constraint W of association A_i should be augmented by the conditions specified in the related mapping fragments (steps (5-6)). Step (7) forms a view 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. If there is more than one minimal covering association, a view is created for each association; the final view is the **UNION** of those views (steps (9-11)).

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 from 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 in ML_D
(7) **let** MP be the string:
"SELECT "+T+ "FROM "+F+"WHERE "+W;
(8) $U \leftarrow U \cup \{MP\}$;
(9) $V \leftarrow$ 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 10: Algorithm for Level View Compilation

In the COMPILE_PC_REL algorithm (Figure 11), views are assigned to parent-child relationships in the CL_D specifying the respective roll up function between child and parent. We add the views that have been already generated for parent and child levels to the join set of the association A_i (steps 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 key attributes are mapped (step (4)). The **WHERE** clause is composed of the join set and possible user defined constraints (steps (7-9)). In steps (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 column of the table to which the such attribute are mapped. The **FROM** part F of a parent-child view is the join set of the minimal covering association.

In COMPILE_FACT_REL (not shown), views are assigned to fact relationships in the CL_D. The main difference with COMPILE_PC_REL is that COMPILE_FACT_REL integrates the views that have been already generated for the bottom levels of each dimension to which the fact relationship is related. Then, for each of these dimension views the algorithm compiles the join set and possible user defined constraints.

Theorem 5.1. *Let S be the number of store tables, F be the number of references between distinct tables and C be the average number of correspondences specified by the user for each construct in the CL_D.The process of compiling all views in our framework is of $O(S + F) + O(C^2 \times F + C^2 \times S \log S) + O(L \times C^2 \log C)$.*

Proof 5.1. *The algorithm generates the SL_D 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 [26]. 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 CL_D.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. With 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.*

```
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 from 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 in ML_D;
(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 11: Algorithm for Parent-child View Compilation

The second step of each run finds 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 [26]. Both are greedy algorithms and run in polynomial time. For example, an implementation of Prim's algorithm that uses the binary heaps and an 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 runs in time $O(C^2)$. Unlike the first step, this step is performed only once in each run of generating a view for each construct in the CL_D.The mapping compilation is executed for all constructs in the CL_D including levels, parent-child relationships and fact relationships. Thus, if L is the number of CL_D 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 $O(S + F) + O(C^2 \times F + C^2 \times S \log S) + O(L \times C^2 \log C)$.

6. Handling Conceptual Level Heterogeneity

As mentioned in Section 4, summarizability is one of the properties a data warehouse has to satisfy to be well-formed. This is not only a requirement for our compilation algorithm but also necessary to efficiently support OLAP queries [24].

However, the user's conceptual view of the data may be non-summarizable, even when the underlying data warehouse is indeed summarizable. In such cases, transforming non-summarizable CL_V dimensions into summarizable ones can be done as a pre-processing mechanism for query evaluation.

A dimension must be both strict and homogeneous to be summarizable. A dimension is *strict* when every element in a level rolls up to no more that one element in each ancestor level. For example, in Figure 3 the CL_V Canadian

Location dimension is strict since we do not expect a branch to belong to two areas, regions, cities, provinces or countries.

A dimension is *homogeneous* when every element in a level rolls up to at least one element in each ancestor level. For example, the CL_V Canadian Location dimension is homogeneous since we expect every branch to belong to some area, region, city, province and country. Putting together both strictness and homogeneity results in every element in a level rolling up to *exactly one* element in each ancestor level. This is the key notion behind the definition of summarizability.

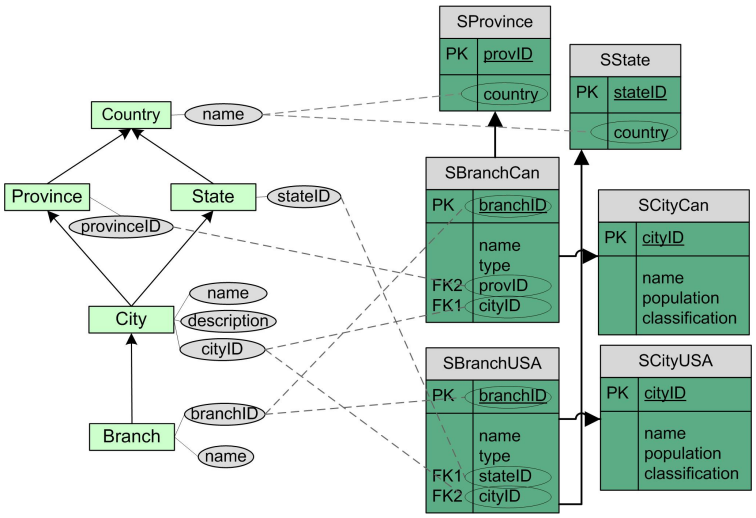


Figure 12: USA and Canada location dimension: heterogeneous CL_V (left) vs. homogeneous SL_V (right)

Example 6.1. *Consider the USA-Canada location dimension for the Dream Home running example depicted in Figure 12. For simplicity, we do not include the area-region path of the motivating example and we only map one attribute per CL_V level. In the figure, every city rolls up to either* Province *(in Canada) or* State *(in the US). (Let us forget for now the case of Washington DC, which rolls up directly to* Country *– we will come back to it later.) As in best practices, the SL_V is well-formed and therefore homogeneous: there are separate tables for branches and cities in the US and Canada to prevent the aggregation problems discussed above. However, the analyst that created the CL_V has decided to represent Canadian and USA branches with the same entity (the same goes for cities); after all, it makes sense from the conceptual point of view to have only one city level in a location dimension.*

The problem with this representation is that the resulting CL_V is non-summarizable, i.e., the roll up functions corresponding to the parent-child relations from City *to* Province *and* State *are not total. Therefore, it is not possible to use the precomputed aggregations at either* State *or* Province *levels to compute the aggregations by country, since the aggregations by state will not include branches in Canada and aggregations by province will not include branches in the US – recall that any single child level should provide all the information for computing OLAP aggregations in the parent level. The analyst expects the system to deal with those query evaluation issues automatically without his intervention.*

A key strategy for speeding up query answering in a multidimensional environment is to reuse precomputed cubes by rewriting queries in terms of them. The process of defining such rewritings is known in the OLAP world as aggregate navigation [6]. The notion of summarizability was originally introduced to be able to use cube views for optimizing query evaluation. In order to compute the aggregated cube for a level, we need to consider every instance of the bottom categories in the aggregation process exactly once. Intuitively, strictness prevents double counting (since every element will be aggregated at most once) and homogeneity prevents under-counting (since no element will be missing during aggregation).

In what follows we propose a methodology for handling heterogeneity. The issues arising from non-strictness have been studied elsewhere (e.g., [29] and [8, Chapter 3]) and are orthogonal to this topic, so we assume strict

dimensions.

6.1. The multidimensional CL_V model

In this section we present the multidimensional model underlying the CL_V. For the definitions below we need to assume 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 $l = \{A_i : V_i \,|\, A_i \in A, V_i \in V, 1 \leq i \leq n\}$.

Definition 6.1. *A CL_V **dimension schema** D is a tuple $(L, \nearrow)$, where L is a set of levels and $\nearrow$ is a binary parent-child relation in $L \times L$, where the first attribute is called the* finer *level and the second one the* coarser *level. We denote by $\nearrow^*$ the transitive and reflexive closure of $\nearrow$. L has a single top level called All reachable from all levels in the dimension, i.e., for each $l \in L$ we have $l \nearrow^* All$. L has a single bottom level l_0 from which all levels in the dimension can be reached, i.e., for each $l \in L$ we have $l_0 \nearrow^* l$.*

In contrast to the model in [24], shortcuts are allowed in a CL_V dimension schema, i.e., for a pair of adjacent levels l and l' in a dimension (i.e., $l \nearrow l'$), there could be an intermediate level l_i such that $l \nearrow^* l_i$ and $l_i \nearrow^* l'$. Other than that, our model follows [24].

Definition 6.2. *A CL_V **dimension instance** I of a dimension schema $D = (L, \nearrow)$ is a tuple $I = (M, R)$, where M is a set of level instances and R is a set of roll up relations. Instances are related to levels by a mapping function $\delta : M \to L$. Let by $ext(l)$ the set of instances mapped to a level l by δ, i.e., $ext(l) := \{x \,|\, \delta(x) = l\}$. For each pair of levels $(l, l') \in L$, $l \nearrow l'$, there exists a roll up relation $RUP^l_{l'}$ in $ext(l) \times ext(l')$. Whenever a roll up relation is functional (i.e., single valued) we denote them by $\mathcal{R}\uparrow^{l'}_{l} : ext(l) \to ext(l')$ to distinguish them from the usual roll up relations.*

Strictness and homogeneity can be defined in terms of the properties of the roll up relations. A dimension is strict if and only if it has only functional roll up relations, i.e., functional roll up relations prevent double counting, which means that every element will be aggregated at most once. A dimension is homogeneous if and only if each roll $RUP^{l'}_{l}$ is total over the elements in l, i.e., total roll up relations prevent under-counting, which means that every element will be aggregated. Consequently, a dimension is summarizable iff every roll up relation is functional and total.

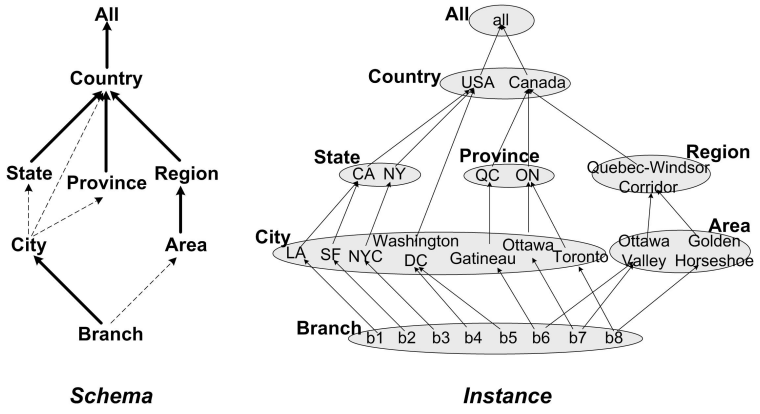


Figure 13: Extended Dream Home dimension schema and instance.

Example 6.2. *Consider the extended Dream Home dimension schema of Figure 13 (left) containing instances of Canada and the US (right). The ovals on the right hand side are the extents of each level identified by the level name, e.g., ext*(Province) = {QC, ON}. *All roll up are functions and are given by the edges on the instance side, e.g., $\mathcal{R}\uparrow^{\text{Province}}_{\text{City}}$= {(Gatineau, QC), (Ottawa, ON), (Toronto, ON)}. Total roll up functions are depicted with bold edges in the schema and partial ones with dashed edges. Since not all edges are in bold, that means the dimension is heterogeneous and hence not summarizable. Looking closer to the instance we can see why this is so: some cities belong to a state, others to a province, and yet another one (*Washington DC*) directly to a country (*USA*). Partial roll up functions prevent*

the use of pre-computed aggregations. For instance, we cannot use the aggregations by State, Province *or* Region *to compute* Country*, since each of the former levels is missing some branches; more importantly, by just looking at the schema we cannot tell exactly which ones.*

Intuitively, the solution of this specific issue is to partition the 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*, as shown in Figure 14. Each of the classes of* Branch *level can be new bottom levels. The schema of these dimensions are also modified according to the data in the bottom level. For example, a level holding the branches of* Washington, DC *does not have either* State *or* Province *levels as its parent level. Note that this new dimension schema has all edges in bold indicating that all roll up functions are total and the new dimension is summarizable.*

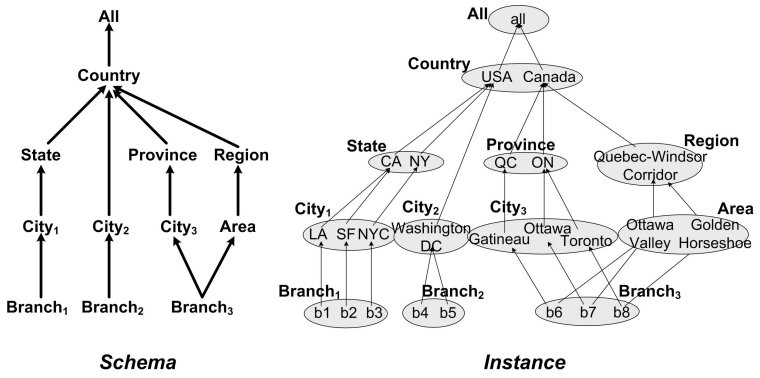


Figure 14: Homogeneous extended Dream Home dimension

If a dimension is summarizable then every level can be aggregated from some other pre-aggregated level(s). The question is which levels are necessary? If a summarizable dimension has a single bottom level, it is easy to see that the aggregation for a level can always be computed from any of its descendants. The reason for this is that having a single bottom level l_0 guarantees that for every level l there is a *total* roll up function $\mathcal{R}\uparrow_{l_0}^{l}$ and therefore no instance of l_0 is missed in the aggregation of l. In contrast, if there are multiple bottom levels, we may need the pre-aggregated information from more than a single level to account for all bottom level instances.

Example 6.3. *Consider again Figure 14. All* City *levels are needed to account for all branches. That means that to aggregate at the* Country *level we could use the pre-aggregations of all three* City *levels confident that all branches are going to be accounted for. Note also that* Area *can be used instead of* City_3 *because the total roll up from* Branch_3 *guarantees that both* Area *and* City_3 *covers all branches in* Branch_3.

The following proposition provides a mechanism to decide whether a given set of levels can be used to aggregate another. Since we are assuming strictness, we will restrict the result to roll up functions, although as originally stated in [24] it applies to the more general case of roll up relations as well.

Proposition 6.1. *A level l is summarizable from a set of levels S in a dimension d if and only if, for every bottom level l_0 of d, we have: $\mathcal{R}\uparrow_{l_0}^{l} = \biguplus_{l_i \in S}(\mathcal{R}\uparrow_{l_0}^{l_i} \circ \mathcal{R}\uparrow_{l_i}^{l})$ where $\biguplus$ is the* additive union*, i.e., union with bag (multiset) semantics.*

Since roll up relations have no duplicates, in order for the equality to hold the additive union must produce the same result as the usual union, i.e., every instance in a bottom level that rolls up to l must pass through exactly one level in S.

The problem of using this proposition to test summarizability from other levels is that it needs the data instances to compute the composition of the roll up functions, i.e., the complexity of this test could be linear in the size of the instances in the candidate S. In addition, there are many candidate S: all possible combinations of descendants of l will need to be tested. Ideally, we would like to decide whether or not a level is summarizable using only the dimension schema. Even better: given a level l, we want to have a schema-based mechanism to actually *find* the set S.

6.2. Making dimensions homogeneous

Heterogeneity in a dimension can be fixed by splitting levels until all roll up functions are total. This cannot be done solely at the schema level: it requires dimension instance information as well. Hurtado et al. [30] propose a framework that models structural irregularities of dimensions by means of integrity constraints on the instances. This class of integrity, 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 dimensions. Using their approach is not feasible in the SCIMM context because conceptual-level users are not data experts and do not have the necessary know-how to specify complex dimension constraints. Therefore, we propose an algorithm for refactoring a dimension into a summarizable one by using the structural information of the dimension schema and instance. To achieve this, we use the well-known notion of *bisimulation* to partition labeled graph nodes into classes based on the structure of the paths they belong to, i.e., two nodes belong to the same class if they are at the end of the same set of label paths from the root. (Given a path p, its label path consists of the sequence of node labels in p.)

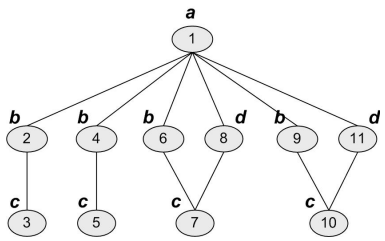


Figure 15: Sample labeled graph

Example 6.4. *Consider the graph of Figure 15, where nodes have labels a, b, c and d. Nodes 3 and 5 are at the end of the same set of label paths from the root,* $\{a.b.c\}$*, so they are bisimilar and belong to the same equivalence class. In contrast, 7 and 10 are at the end of a different set of label paths,* $\{a.b.c, a.d.c\}$*, and thus they belong to a different class. All nodes labeled by b belong to the same class because they are all at the end of a.b. Similarly with nodes labeled by d, they are all bisimilar.*

The widespread use of bisimulation to compute this kind of partition is motivated by its relatively low computational complexity properties: the partition can be computed in time $O(m \log n)$ (where m is the number of edges and n is the number of nodes in a labeled graph) as shown in [31], or even linearly for acyclic graphs, as shown in [32].

Bisimulation has been used in XML and other graph-structured data to create summaries (i.e., synopsis) of large heterogeneous data graphs (see [33, Chapter 4] for a recent survey). To the best of our knowledge, bisimulation has never been used in the context of OLAP and data warehouses.

6.3. Bisimulation

In this section, we briefly explain the notion of bisimulation, which can serve as a means of generating summarizable subdimensions. Bisimulation defines similarity among nodes based on the label paths the nodes belong to.

The instance $I = (M, R)$ of a dimension schema $D = (L, \nearrow)$ can be viewed as a directed labeled graph $G = \langle M, E, L, \delta \rangle$ where M is the set of nodes with a distinguished sink node denoted *all*, E is the set of edges, L is the set of labels, the mapping function $\delta : M \to L$ works as a labeling function that assigns labels (i.e., levels) to nodes (i.e., instances), and there is an edge (m, m') in E if and only if $\exists\, l, l'$ such that $l \nearrow l'$ and $\mathcal{R}\uparrow_l^{l'}(m) = m'$.

Example 6.5. *Figure 16 shows the instance graph of Branch Location in Figures 13 and 14. In this graph, one node is defined for each element of the instance; the nodes are labeled by the name of the level to which they belong. The parent-child relation between two elements is reflected by an edge drawn from the child element to the parent one.*

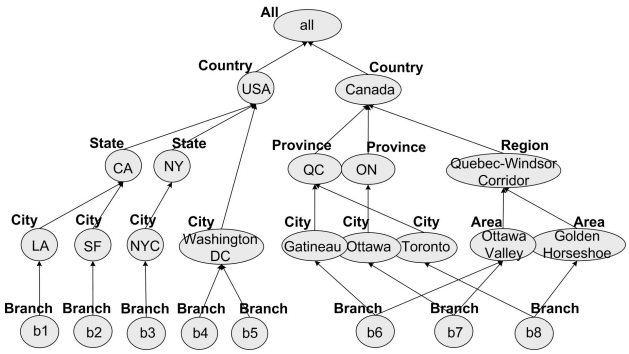


Figure 16: The instance graph of the dimension in Figures 13 and 14.

Definition 6.3 (bisimulation and bisimilarity). *Let $G = \langle M, E, L, \delta \rangle$ be an instance graph. A symmetric, binary relation $\approx$ on G is called a* bisimulation *if for any two nodes x and y in M, with $x \approx y$, we have that (i) $\delta(x) = \delta(y)$ and (ii) if x' is the parent of x (i.e., $(x, x') \in E$) and y' is the parent of y (i.e., $(y, y') \in E$) then $x' \approx y'$. Two nodes x and y in G are said to be* bisimilar*, denoted by $x \approx^b y$, if there is some* bisimulation $\approx$ *such that $x \approx y$.*

Given an instance graph $G = \langle M, E, L, \delta \rangle$, a bisimulation $\approx$ induces a partition $\mathbb{P} = \{P_1, \ldots, P_n\}$ of M where each set has the same label (by definition of bisimulation); by slightly abuse of notation we denote by $\delta(P_i)$ the label shared by all nodes in P_i. From G and $\mathbb{P}$ we create a *refactored* dimension with schema $(L, \nearrow)$ and instance (M, R) as follows: (a) there is level $l_i \in L$ with $ext(l_i) = P_i$ iff there is a $P_i \in \mathbb{P}$ and $\delta(P_i) = l_i$, (b) for each pair of levels $l, l' \in L$, we have that $l \nearrow l'$ iff for every $x \in ext(l)$ there exists $x' \in ext(l')$ such that $(x, x') \in E$, (c) $\mathcal{R}\uparrow_l^{l'}(x) = x'$ iff $(x, x') \in E$. A refactored dimension is equivalent to a bisimilarity graph [34].

Example 6.6. *Figure 14 shows the refactored dimension corresponding to that instance graph in Figure 16. The partition induced by bisimulation consists of ovals on the right hand side of the former figure. For instance, nodes b_1, b_2, and b_3 are at the end of label path* All.Country.State.City.Branch *and constitute the extent of* Branch_1. *We use subindexes in the figure for identification purposes, but they are not part of the labels, i.e.,* Branch_1, Branch_2 *and* Branch_3 *are all considered to have the same label* Branch. *There is an edge between* Branch_1 *and* City_1 *in the schema because b_1, b_2, and b_3 in* Branch_1 *have edges to nodes in* City_1. *Finally,* $\mathcal{R}\uparrow^{\mathsf{City}_1}_{\mathsf{Branch}_1}(b_1) = \mathsf{LA}$, $\mathcal{R}\uparrow^{\mathsf{City}_1}_{\mathsf{Branch}_1}(b_2) = \mathsf{SF}$, *and* $\mathcal{R}\uparrow^{\mathsf{City}_1}_{\mathsf{Branch}_1}(b_3) = \mathsf{NYC}$.

The issue now is how to use the refactored dimension for aggregation, for which we defined the following proposition

Proposition 6.2. *To compute the aggregations of a level l in a refactored dimension $D = (L, \nearrow)$ we can use the precomputed aggregations of a set of levels $S := \{l_1, \ldots, l_n\}$, $S \subset L$, such that S satisfies the following two conditions: (a) $l_i \in S$ belongs to* some path *connecting l to a bottom level, and (b) not two levels in S belong to paths connecting l to* the same *bottom level.*

The intuition behind Proposition 6.2 is simple. If there were more than one level $l_i \in S$ per bottom level, some instances would be counted twice. If there were no $l_i \in S$ for some bottom level, some instances would be missed. The only way of having the right aggregation in any given level is to count the bottom level instances exactly one, which is what the proposition guarantees.

Note that, given that all roll up functions in a refactored dimension are total, Proposition 6.2 is equivalent to Proposition 6.1 and we can therefore say that level l is summarizable from S. The next example illustrates the use of Proposition 6.2 for computing aggregations on the Figure 14 dimension.

Example 6.7. *Let us consider the* Country *level in Figure 14. There are four different paths from* Country *to the three bottom levels on the schema, namely* $\mathsf{State}\text{-}\mathsf{City}_1.\mathsf{Branch}_1$, $\mathsf{City}_2.\mathsf{Branch}_2$, $\mathsf{Province}.\mathsf{City}_2.\mathsf{Branch}_2$ *and* $\mathsf{Region}.\mathsf{Area}.\mathsf{Branch}_2$. *For computing the Country aggregation, we could take all bottom levels and make* $S = \{ \mathsf{Branch}_1, \mathsf{Branch}_2, \mathsf{Branch}_3 \}$, *which guarantees that every branch will be counted exactly once. But we could also use the aggregations of higher*

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

Table 1: Benchmark parameters

levels in the dimension, as long as the chosen set of levels satisfy Proposition 6.2. For instance, we could use $S' = \{$ City_1, City_2, City_3 $\}$. *Note that by using* S', *as in the previous case with* S, *all branches in the bottom levels will be counted exactly once because each roll up function involved is total. Replacing* City_3 *by* Area *or* Region *in* S' *will also compute the correct aggregation -* Area *and* Region *cover the same branches that* City_3.

7. Experimental Evaluation

7.1. Setup

We implemented our mapping compilation algorithm (Section 5) in Java using MySQL [1] for physical storage, the Codehaus streaming API for XML, StAX [2] to parse XML, and the Annas [3] graph and algorithm package for efficient implementations of the Dijkstra and Prime algorithms. Each SL_V model is generated from the MySQL schemas of the tested physical models. The output of the mapping compilation algorithm is a 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 CL_V, (2) number of mapping fragments in the ML_V, (3) number of tables in the SL_V, and (4) "density" of referential constraints in the store schema (i.e., number of foreign keys between tables).

The experiments were performed on three multidimensional schemas: the TPC-H Benchmark[4], the Dream Home case study [3] from Section 1 and an Infection Control Data Warehouse (in use at the Ottawa Hospital) [35], a conceptual model showing information about patients' admission, activities, and discharge in order to monitor and reduce the risk of infections. For each schema, we varied the above parameters.

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 a Windows XP Pentium(R)D 3.40 GHz machine with 2 GB of RAM. Each result is the average of 500 runs. We varied the number of SL_V tables, the density of the foreign keys, and the number of mapping fragments per CL_V level. For each CL_V level, we select a random number of tables from the SL_V to map to. The number of tables to which an attribute can be mapped is also selected randomly.

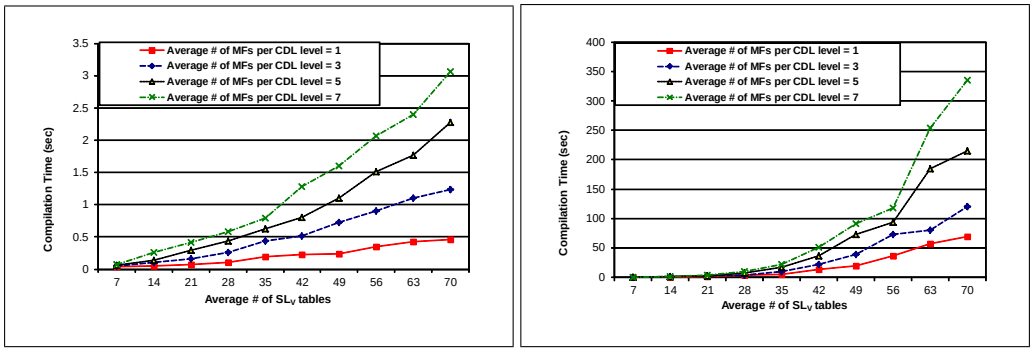
We tested two types of SL_V schemas: *well-formed* and *fully-connected*. The well-formed schemas satisfy Definition 4.3. In fully-connected schemas, each SL_V table has foreign keys to all other tables in the schema. The fully-connected schemas are not well-formed; however, they allow us to study the performance of our algorithm in the presence of unusually high numbers of referential constraints. As previously discussed, our algorithm is only guaranteed to return exact views for well-formed schemas. The views generated for fully-connected schemas are guaranteed to be 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 1 respectively summarize the range of sizes of the CL_V, ML_V and SL_V tested in each domain. Size is measured by the number of levels, parent-child relationships and fact relationships for the CL_V, mapping fragments for the ML_V, and tables for the SL_V. The fourth and fifth rows report the number of foreign keys for

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

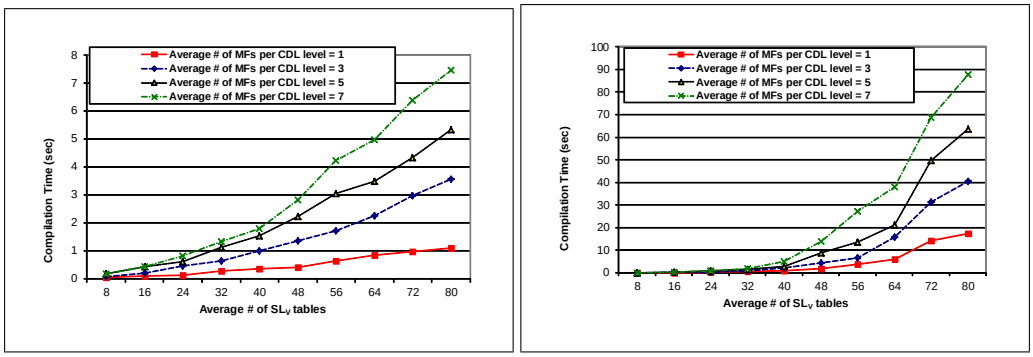
the fully-connected and the well-formed schemas, respectively. The sixth row shows the number of views generated during mapping compilation in each case.

7.2. Experimental Results



(a) Well-formed SL_V Schema (b) Fully-connected SL_V Schema

Figure 17: Compilation Time vs. Average # of SL_V tables for TPC-H



(a) Well-formed SL_V Schema (b) Fully-connected SL_V Schema

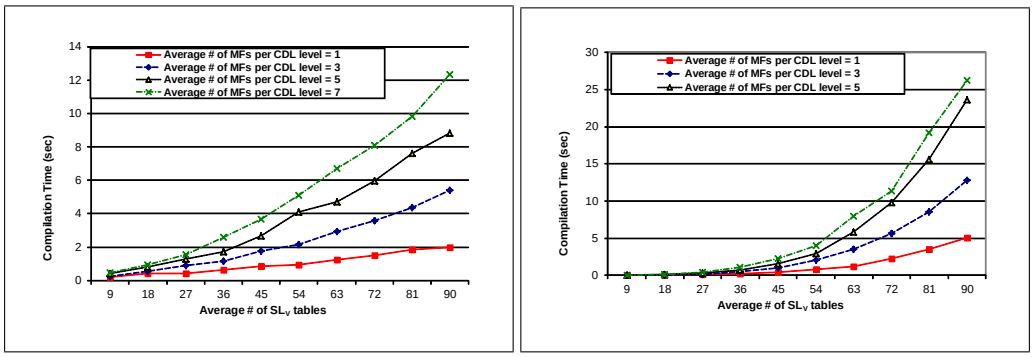
Figure 18: Compilation Time vs. Average # of SL_V tables for Dream Home

The first set of experiments evaluates the performance of the compilation algorithm for well-formed schemas (Figures 17(a), 18(a) and 19(a)) with a fixed CL_V schema and varying the number of SL_V elements and mapping fragments. These figures depict the time to generate views for all constructs in each conceptual model versus the average number of SL_V 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 SL_V.

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 SL_V elements and mapping fragments. However, this time the SL_V 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 17(b), 18(b) and 19(b).

Our results suggest that the mapping compilation algorithm performs better in practice than its worst-case complexity. Consider the curve corresponding to 5 mapping fragments per CL_V level on TPC-H with a well-formed schema (Figure 17(a)). 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 ~ 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 6. Hence the first set of experiments fairly accurately represent mapping compilation in real data



(a) Well-formed SL_V Schema (b) Fully-connected SL_V Schema

Figure 19: Compilation Time vs. Average # of SL_V tables for Infection Control

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

8. Conclusion

This paper presented a SCIMM Framework in which the relationships between the conceptual model and multidimensional data are specified by users using simple attribute-to-attribute correspondences.

We proposed a polynomial time algorithm for compiling such correspondences into mappings that associate each construct in the conceptual model with a view on the physical model. These compiled views are sound in all cases and exact when the physical data warehouses are well-formed. We discussed experimental results showing that our algorithm works well in practice, specially when dealing with well-formed data warehouses.

We are considering extending this work in several directions, the first of which is query evaluation. 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. Another interesting direction is to investigate ways for integrating the compilation algorithm with the dimension refactoring process for making a dimension summarizable (and hence the data warehouse well-formed). This way we could always get exact views without the need of changing the physical structure of the data warehouse.

Acknowledgements

This work is partially funded by grants from NSERC Canada and SAP.

[1] Data, data everywhere: A special report on managing information, The Economist 2010 (February 27).
[2] J. A. Blakeley, S. Muralidhar, A. Nori, The ADO.NET entity framework: Making the conceptual level real, in: ER, 2006, pp. 552–565.
[3] T. Connolly, C. Begg, Database Systems: A Practical Approach to Design, Implementation, and Management, Addison Wesley, 2009.
[4] F. Rizzolo, I. Kiringa, R. Pottinger, K. Wong, The conceptual integration modeling framework: Abstracting from the multidimensional model, CoRR arXiv:1009.0255 (2010) 1–18.
[5] J.-N. Mazón, J. Lechtenbörger, J. Trujillo, A survey on summarizability issues in multidimensional modeling, DKE 68 (12) (2009) 1452–1469.
[6] R. Kimball, L. Reeves, M. Ross, W. Thornwaite, The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses, John Wiley, 1998.
[7] R. Torlone, Conceptual multidimensional models, in: Multidimensional Databases, 2003, pp. 69–90.
[8] E. Malinowski, E. Zimányi, Advanced Data Warehouse Design: From Coventional to Spatial and Temporal Applications, Springer, 2008.
[9] P. Giorgini, S. Rizzi, M. Garzetti, Grand: A goal-oriented approach to requirement analysis in data warehouses, Decision Support Systems 45 (1) (2008) 4–21.

[10] J.-N. Mazón, J. Trujillo, J. Lechtenbörger, Reconciling requirement-driven data warehouses with data sources via multidimensional normal forms, Data Knowl. Eng. 63 (3) (2007) 725–751.
[11] J.-N. Mazón, J. Trujillo, A hybrid model driven development framework for the multidimensional modeling of data warehouses!, SIGMOD Record 38 (2) (2009) 12–17.
[12] O. Romero, A. Abelló, Automatic validation of requirements to support multidimensional design, Data Knowl. Eng. 69 (9) (2010) 917–942.
[13] O. Romero, A. Abelló, A framework for multidimensional design of data warehouses from ontologies, Data Knowl. Eng. 69 (11) (2010) 1138–1157.
[14] K. Hahn, C. Sapia, M. Blaschka, Automatically generating olap schemata from conceptual graphical models, in: DOLAP, 2000.
[15] P. P. Chen, The entity-relationship model - toward a unified view of data, ACM Trans. Database Syst. 1 (1) (1976) 9–36.
[16] A. Adya, J. A. Blakeley, S. Melnik, S. Muralidhar, Anatomy of the ado.net entity framework, in: SIGMOD, 2007, pp. 877–888.
[17] Hibernate project, `http://www.hibernate.org/`.
[18] C. Howson, BusinessObjects XI (Release 2): The Complete Reference, McGraw-Hill, 2006.
[19] D. Volitich, IBM Cognos 8 Business Intelligence: The Official Guide, McGraw-Hill, 2008.
[20] M. Lenzerini, Data integration: A theoretical perspective, in: PODS, 2002, pp. 233–246.
[21] P. G. Kolaitis, Schema mappings, data exchange, and metadata management, in: PODS, 2005, pp. 61–75.
[22] A. Doan, A. Y. Halevy, Semantic integration research in the database community: A brief survey, AI Magazine 26 (1) (2005) 83–94.
[23] M. Levene, G. Loizou, Why is the snowflake schema a good data warehouse design?, Information Systems (2003) 225 – 240.
[24] C. A. Hurtado, C. Gutiérrez, A. O. Mendelzon, Capturing summarizability with integrity constraints in olap, TODS 30 (3) (2005) 854–886.
[25] T. B. Pedersen, C. S. Jensen, C. E. Dyreson, A foundation for capturing and querying complex multidimensional data, Inf. Syst. 26 (5).
[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, MIT Press, 2001.
[27] D. Maier, A. O. Mendelzon, Y. Sagiv, Testing implications of data dependencies, ACM Trans. Database Syst. 4 (4) (1979) 455–469.
[28] D. S. Johnson, A. Klug, Testing containment of conjunctive queries under functional and inclusion dependencies, in: PODS, 1982.
[29] T. B. Pedersen, C. S. Jensen, C. E. Dyreson, Extending practical pre-aggregation in on-line analytical processing, in: VLDB, 1999.
[30] C. Hurtado, C. Gutierrez, Computing cube view dependences in olap datacubes, SSDBM (2003) 33–42.
[31] R. Paige, R. E. Tarjan, Three partition refinement algorithms, SIAM J. Comput. 16 (6) (1987) 973–989.
[32] A. Dovier, C. Piazza, A. Policriti, An efficient algorithm for computing bisimulation equivalence, TCC 311 (1-3) (2004) 221–256.
[33] F. Rizzolo, DescribeX: A framework for exploring and querying XML Web collections, Ph.D. thesis, University of Toronto (2008).
[34] R. Kaushik, P. Bohannon, J. F. Naughton, H. F. Korth, Covering indexes for branching path queries, in: SIGMOD, 2002, pp. 133–144.
[35] B. Eze, C. Kuziemsky, L. Peyton, G. Middleton, A. Mouttham, Policy-based data integration for e-health monitoring processes in a B2B environment: experiences from Canada, J. Theor. Appl. Electron. Commer. Res. 5 (1) (2010) 56–70.