Specifying OLAP Cubes On XML Data

Authors: Mikael R. Jensen
Thomas H. Møller
Torben Bach Pedersen

Technical Report 01-5003
Department of Computer Science
Aalborg University

Created on June 13th, 2001

Specifying OLAP Cubes On XML Data

Mikael R. Jensen Thomas H. Møller Torben Bach Pedersen
Department of Computer Science, Aalborg University
{mrj,thm,tbp}@cs.auc.dk

Abstract

On-Line Analytical Processing (OLAP) enables analysts to gain insight into data through fast and interactive access to a variety of possible views on information, organized in a *dimensional* model. The demand for data integration is rapidly becoming larger as more and more information sources appear in modern enterprises. In the data warehousing approach, selected information is extracted in advance and stored in a repository. This approach is used because of its high performance. However, in many situations a logical (rather than physical) integration of data is preferable. Previous web-based data integration efforts have focused almost exclusively on the logical level of data models, creating a need for techniques focused on the conceptual level. Also, previous integration techniques for web-based data have not addresses the special needs of OLAP tools such as handling dimensions with hierarchies.

Extensible Markup Language (XML) is fast becoming the new standard for data representation and exchange on the World Wide Web. The rapid emergence of XML data on the web, e.g., business-to-business (B2B) e-commerce, is making it necessary for OLAP and other data analysis tools to handle XML data as well as traditional data formats..

Based on a real-world case study, this paper presents an approach to the conceptual specification of OLAP DBs based on web data. Unlike previous work, this approach takes special OLAP issues such as dimension hierarchies and correct aggregation of data into account. Additionally, an integration architecture that allows the logical integration of XML and relational data sources for use by OLAP tools is presented.

1 Introduction

On-Line Analytical Processing (OLAP) is a category of business software tools that enables decision support based on multidimensional analysis of summary data. OLAP data, typically drawn from a physical integration of transactional databases, is organized in *multidimensional data models*, categorizing data as either measurable *facts* (measures) or hierarchically organized *dimensions* characterizing the facts. Features like automatic aggregation [28] and visual querying [33] supported by OLAP tools ease the process of decision support compared to traditional database management systems.

Integration of distributed data sources is becoming increasingly important as more business relevant data appear on the web, e.g., on B2B marketplaces, and enterprises cooperate more tightly with their partners, creating a need for integrating the information from several enterprises. The *data warehousing* approach dictates a physical integration of data, mapping data from different information sources into a common multidimensional database schema. This enables fast evaluation of complex queries, but demands great effort in keeping the data warehouse up to date, e.g. when data passes from the sources of the application-oriented operational environment to the data warehouse, inconsistencies and redundancies must be resolved, so the data warehouse provides an integrated and reconciled view of the data of the organization. However, some kinds of data are impossible, or not attractive, to physically store in a data warehouse, either because of legislation or because of the nature of the data, e.g., rapidly dynamically evolving information. Enabling integrated use of such data requires a logical rather than physical integration.

XML is a meta language used to describe the structure and content of documents. XML, although originally a document markup language, is increasingly used for data exchange on the Web. The application of XML as a standard exchange format for data available on the Web makes it attractive to use in conjunction with OLAP tools.

Previous approaches for integrating web-based data, particularly in XML format, have focused almost exclusively on data integration at the *logical* level of the data model, creating a need for techniques that are usable at the conceptual level which is more suitable for use by system designers and end users. The most wide-spread conceptual model is the Unified Modeling Language (UML) [23] which is used in this paper. Previous integration approaches have not addressed the special issues related to OLAP, e.g., dimensions with hierarchies, and are thus not optimally suited for integrating data to be used for OLAP.

The contribution of this paper is threefold. First, an architecture for integrating XML data at the conceptual level is presented. The architecture also supports relational data sources and is thus well suited to build data warehouses which are based partly on in-house relational data and partly on XML data available on the web. A prototype system supporting the architecture is currently being developed as part of a project that investigates technologies for building data warehouses based on web data. Although this paper primarily argues for a logical data integration, the specification techniques may also be used for physical data integration, i.e., data warehousing. Second, we propose a multidimensional model, the UML snowflake diagram, enabling a precise specification of an OLAP DB based on multiple XML and/or relational data sources. The UML diagramming method is used for describing and visualizing the logical structure of XML documents, easing the design of the OLAP DB. Third, we describe how to handle the special considerations that need to be taken when designing an OLAP DB on top of XML data, such as dimensions with hierarchies and ensuring correct aggregation of data.

We believe this paper to be the first to address the issue of integrating XML and/or relational data for use by OLAP tools and to describe the special considerations that need to be taken in this context. Also, we believe this paper to be the first to consider the integration of XML and relational data at the conceptual rather than the logical level.

A number of systems allows integration of structured and/or semi-structured data. YAT [2], Strudel [9], TSIMMIS [12], and Ozone [19] all use a semi-structured/XML data model, Garlic [29] uses an object data model, while Cohera [14] uses the relational (SQL) data model. These systems all focus on web data integration at the *logical* level. In comparison, the approach described in this paper takes special OLAP characteristics such as fact tables, dimensions with hierarchies, and measures into account. Also, this paper focus on integration of web data on the *conceptual* level, supporting the *design* process better. However, we do not consider data integration at the logical level to be unnecessary, as such systems are used for the *implementation* of the integrated databases. Indeed, a DW conceptually designed using our approach could be implemented using these systems. The issue of converting XML data into relational data is described in [8, 10, 30]. In comparison, this paper aims at capturing more semantics of the XML data by using a higher-level conceptual model. Also, the data structures obtained by our approach is better suited to OLAP analysis as important DW/OLAP issues such as fact tables, dimensions with hierarchies, and measures are addressed directly.

The remainder of the paper is organized as follows. Section 2 presents a case study concerning B2B e-commerce of electronic components. Section 3 presents an architectural overview describing how XML and relational data can be integrated and used for OLAP analysis. Section 4 defines a multidimensional model capturing the structure and content of a logical specification of OLAP DBs on top of XML and relational data sources, and describes the specific issues related to designing an OLAP DB on XML data. Section 5 summarizes and points to topics for future research. Appendices A, B, and C provide additional details for the interested reader but the paper can be understood wihtout reading them.

2 Motivation—A Case Study

This section describes a case study motivating the need for integrating various XML documents and relational data sources into a common multidimensional DB, serving as a basis for data analysis in an OLAP system. The case study is a simplification of the Electronic Component Information Exchange QuickData Architecture (ECIX) [31]. The ECIX QuickData Architecture is a project dedicated to the design of standards for B2B technical information exchange of component information for traditional electronic components. The ECIX standards defines standards for both XML data exchange as well as architectures for Web-enabled systems for information exchange between all the participants involved in the design and manufacture of electronic systems. For a thorough description of the

ECIX QuickData Architecture, see [31]. The case study concerns retailers and suppliers of electronic components. The retailer sells electronic components to end users bought from a number of different suppliers. The retailer stores information about component sales and customers as both XML and relational data, and the suppliers provide basic information about their products as XML data on the Web. The goal is for the retailer to construct an OLAP DB based on the data stored by the retailer as well as the XML data provided by the suppliers, in order to enable analysis of component sales.

An XML document has both structure and content, and XML provides a means for separating one from the other in the electronic document. The *structure* of a document is given by use of matching tag pairs (termed an element) and the information between matching tags is referred to as the *content* of the element. Furthermore, an element is permitted to have additional attributes, where values are assigned to the attributes in the start tag of the element. XML documents can also contain a description of their logical structure, which is called a document type definition (DTD). A DTD is a context free grammar defining, in terms of element content specifications, all allowable elements, their attributes, and the elements nesting structure. Given a DTD it can be verified that an XML document *conforms* to the DTD, and if so, the XML document is said to be *valid*. For a complete specification and description of XML, see [34, 27].

The data stored by the *retailer* is an XML document describing sales, an XML document describing the component numbering systems used by the suppliers, and a relational database containing information about customers.

Sales Document The sales document describes each sale made by the retailer. Each sale consists of a unique sales id, a date, a customer id, and a list of items sold. The DTD describing the structure and contents of the sales document is shown below. Every sales document conforms to this DTD.

```
<?xml version="1.0" encoding="utf-8" ?>
<!ELEMENT salesDB (sales*) >
<!ELEMENT sales (date, customerID, item+) >
<!ATTLIST sales salesID ID #REQUIRED >
<!ELEMENT date (#PCDATA) >
<!ELEMENT customerID (#PCDATA) >
<!ELEMENT item (price, componentID) >
<!ELEMENT price (#PCDATA) >
<!ELEMENT componentID (#PCDATA) >
```

An example of an XML document conforming to the above DTD is shown below. The document describes sale no. s48372 at the 2nd of October 2000 to the customer having id 2347. The customer bought the package of components having id r12200 for a price of $6.25.

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE salesDB SYSTEM "sales.dtd">
<salesDB>
  <sales salesID="s48372">
    <date> 2000-10-02 </date>
    <customerID> 2347 </customerID>
    <item>
      <price> 6.25 </price>
      <componentID> r12200 </componentID>
    </item>
    ...
  </sales>
  ...
</salesDB>
```

Component Numbering Document The component numbering system used by the retailer is different from the ones used by the suppliers. In order to map between the two numbering systems, the retailer has constructed an XML document providing information about each supplier as well as a mapping between the numbering systems used by the retailer and the supplier. The mapping document describes the name, address, phone, the URL of each

supplier, and a list of component ids mapping between the supplier's component ids and the ids used by the retailer. The DTD describing the structure and contents of the mapping document and an example XML mapping document can be seen in Appendix A.

Relational Customer Database The retailer stores information about customers in a relational database. For each customer, data is stored stating the id of the customer, the name, the address (street and number), the city, and the country in which the customer lives. The schema of the table is shown below.

```
customer(id, name, street, number, city, country)
```

A tuple from an instance of this relation is shown below.

```
(2347, 'Peter Scott', 'Lambda Lane', '6', 'London', 'England')
```

The XML data provided by each component *supplier* is a document describing basic properties of the different products sold by the supplier. The properties described for each component are properties such as the name of the class to which the component belongs, the name and id of the component, the number of pins and gates, an optional textual description, and the price and number of components in one package. Furthermore, since some suppliers also sell devices built from different components, the component document contain information about these devices, as well as a reference from components to devices indicating that a particular component is used to build a particular device. The DTD describing the structure and contents of the component document and an example XML component document can be seen in Appendix A.

3 Integration Architecture

This section describes a general system architecture for integrating XML and relational data sources at the conceptual level in a web-based OLAP DB. The architecture makes the use of XML data transparent to the data analysis tool and thereby to the end user.

XML Data Sources Before the designer can specify an OLAP based on XML data, the designer needs to know the structure of the data to be used. One way of obtaining this knowledge is to fetch the entire XML document from the Web. This solution is rather unsatisfactory, since fetching the entire XML document can take a considerable amount of time. Another solution is to consider the logical structure of the XML document, described by the DTD. Fetching only the DTD saves time, since the DTD is small in size compared to the document it describes. A potential problem associated with this approach is that the DTD can be overly complex, giving no contribution to the designer's understanding of the structure of the XML data, a problem which is worsened by the textual format of the DTDs. What is needed is an easily comprehendible and quick way of communicating the structure of an arbitrary XML data source to the designer.

To accomplish this goal, the DTD is transformed into a Unified Modeling Language (UML) diagram. Visualizing the structure of XML documents in a graphical way makes it easier for the designer to understand than a context free grammar such as the DTD. By describing the structure of XML data sources using a high-level conceptual graphical modeling language, each XML data source on the Web becomes an easy accessible and comprehensible database to both designers and end users. The use of DTDs as a basis for deriving the graphical representation of the data source supports fast browsing of the available data, due to the small size of the DTD. This corresponds to the traditional use of the Web where browsing is probably the most important mean of finding the desired information. A detailed specification of the transformation from DTDs to UML diagrams is given in another paper [16].

It is assumed throughout this paper that the logical structure of an XML document is described by a DTD and that the XML document is valid. Notice, that other formalisms for describing XML document structure, more powerful than the DTD formalism, exist, e.g. XML Schema [35], but the DTD is the only formalism that is a World Wide Web Consortium (W3C) Recommendation. The XML 1.0 Recommendation [34] is used throughout this paper.

Relational Data Sources An integration of XML and relational data is necessary since an enterprise typically stores different types of transactional data in relational databases. As is the case for XML data, a conceptual model of the structure of the relational data is needed, in order for the designer to make qualified statements about the data to be used. It is not the focus of this paper to describe conceptual models for relational data and it is therefore assumed that UML diagrams describing the relational data can be made available to the designer. Otherwise, database design tools such as ERwin [7] can aid in the process by doing reverse engineering of relational schemas to UML diagrams. We have chosen to integrate the relational data directly, rather than mapping it into XML first, as such a translation would mean that some of the semantics of the data, most likely important to the OLAP DB, would get lost.

General System Architecture Figure 1 illustrates the architecture which enables the use of XML data and/or relational data in a "virtual" OLAP DB for performing decision support using existing OLAP tools.

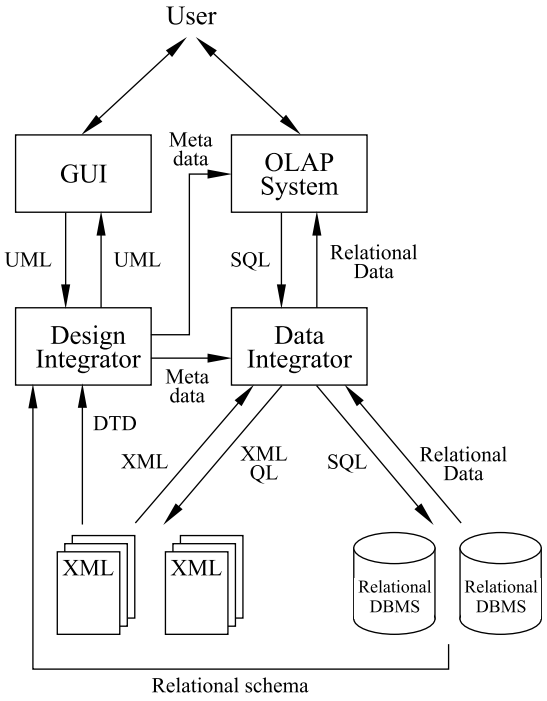

Figure 1: Architecture

The first part of the data integration process is the conceptual integration based on the UML diagrams, resulting in a conceptual model of the OLAP DB. The OLAP DB model (termed a UML snowflake diagram) is specified by the designer through a graphical user interface (GUI). The UML snowflake diagram is constructed from UML diagrams describing the data sources, these diagrams are generated by the Design Integrator. The construction of a UML snowflake diagram is described in detail in Section 4. The UML snowflake diagram is transformed into relational structures by the Data Integrator, and are then made available to the OLAP System. Metadata about the specification of the OLAP DB such as measures, default aggregation function, fact tables, dimension tables, and join keys are transferred directly into the OLAP System, meaning that no separate design effort is needed here.

At run-time, the data sources are presented to the OLAP tools as relations, creating a relational multidimensional view (a star or snowflake schema [18] as these are better suited for OLAP analysis) over the data sources. There are several good reasons for not using UML as the data model for the run-time system. First, no query language for UML exists, so no query tools would be able to use the constructed OLAP DB. Second, almost all OLAP tools can access data using SQL, while only very few, if any, OLAP tools can use XML query languages for data access. This makes the relational model the obvious choice for the run-time system. The central component in the run-time systems is the Data Integrator which enables both XML data and relational data to be accessed using SQL queries through a standard interface like Open Database Connectivity (ODBC). Integrating XML this way makes the use of XML transparent to the end user since the OLAP tool cannot tell the difference between "true" relational structures

and relational structures generated from XML data. The relational OLAP (ROLAP) approach has been chosen over Multidimensional OLAP (MOLAP) as ROLAP has the widest choice of analysis tools since both ROLAP and most MOLAP tools as well as standard relational query tools can access ROLAP data sources.

At query time, the end user query the OLAP System using an OLAP query language native to the OLAP System, e.g. MultiDimensional eXpression (MDX) as used by Microsoft's SQL Server 2000 Analysis Services [22]. These queries are transformed into one or more SQL queries by the OLAP System, used to query the relational structures generated from the UML Snowflake Diagram. The Data Integrator examines the SQL queries delivered by the OLAP System and transforms these queries into a series of queries expressed in either an XML query language or SQL, depending on whether the data originates from an XML source or a relational database management system (RDBMS). The results of these transformed queries (XML from XML data sources and relational tuples from RDBMSs) are then transformed into relational tuples fitting the relational schema generated from the UML snowflake diagram. A detailed description of this query translation is beyond the scope of this paper.

4 Specifying The OLAP Cube

This section describes the specification of an OLAP DB on XML documents and/or relational data sources. The constructed diagram, termed a UML snowflake diagram, resembles the structure of the relational snowflake schema, but is composed entirely of UML classes. The UML snowflake diagram constitutes the schema of an entire OLAP DB. The use of UML as a conceptual modeling language for the OLAP DB makes the integration of elements from different data sources easier, since the data sources, both XML and relational, are also described by UML diagrams (as assumed in Section 3). Furthermore, the UML model is a high-level semantic data model with a graphical notation, making it convenient to illustrate relationships between data elements.

The construction of a UML snowflake diagram is not a fully automatic process and hence relies heavily on the designer's knowledge about the domain. However, when based on XML data, some structural properties inherent to XML aids the design process. More specifically, XML specifics, e.g., hierarchical nesting of data, are made into a number of *design patterns* [11] in the design tool that the designer can then choose from in order to specify the OLAP DB more easily.

We now formally define a UML snowflake diagram and a set of rules defining how to construct a UML snowflake diagram consisting of data elements from various data sources. The possibility of mixing data elements from various data sources enables the use of XML data as a foundation for a complete OLAP DB, or as part of fact or dimensional data only. Although this paper argues for a logical rather than physical integration of data, the constructed UML snowflake diagram can equally well be used as a foundation for a physical integration of the data elements into a traditional data warehouse [17, 18].

In order to precisely specify properties of the UML snowflake diagram, a language for querying UML diagrams is needed. As UML does not have a query language of it's own, we are forced to use an alternative. However, for the subset of UML that we have chosen, we can specify a precise, bijective mapping between UML diagrams and the ODMG data model [5], enabling us to use the OQL query language. The precise definition of this mapping can be seen in Appendix B.

4.1 UML Snowflake Diagram

We now define the UML snowflake diagram which is a specification of an OLAP DB constructed from UML classes. The snowflake diagram can be composed of elements from different UML diagrams, thereby describing the integration of several possibly heterogeneous data sources.

Definition 4.1 UML Snowflake Diagram
A UML snowflake diagram is a rooted graph consisting of UML classes satisfying the following conditions:

1. The edge from *Root* (the designated root-class) to all classes having a path of length $n = 1$ from *Root* is an association type link with cardinality 0..* and 1

2. The edge between *class*$_i$ having a path of length $n \geq 1$ from *Root* and *class*$_j$ having a path of length $m = n+1$ from *Root* is an aggregation type link with cardinality 1 and 1..*, from *class*$_j$ to *class*$_i$

3. A class *class*$_i$ connected to *Root* through an association type link along with all classes having a path of aggregation type links of length $n \geq 1$ from *class*$_i$ is termed a *subgraph*. No edges exists between subgraphs. Note that subparts of dimension hierarchies are *not* subgraphs.

4. No edges exist between *class*$_i$ and *class*$_j$ within a subgraph if the length of the paths from *Root* to *class*$_i$ and *class*$_j$ are equal.

The class *Root* in the UML snowflake diagram is termed a *fact class*. Each subgraph is termed a *dimension* to *Root* and each path *class*$_1$, *class*$_2$, ..., *class*$_n$ from root to leaf in the subgraph is termed a *hierarchy*. Classes *class*$_1$, *class*$_2$, ..., *class*$_n$ in the path each define a *hierarchy level* for the dimension. The connection between *Root* and *class*$_1$ is established through an association with cardinality 0..* and 1, whereas *class*$_i$ is an aggregation of *class*$_j$ ($i < j > 1$) with cardinality 1 and 1..*. Aggregations between hierarchy levels in a dimension are used to explicitly express the "consist of" relationship normally used in UML diagrams. This convention is consistent with the transformation from DTDs to UML diagrams described in [16], where the nesting structures of XML documents are modeled as aggregations. Associations between the fact class and the dimensions are used since no "consist of" relationship exist between fact and dimensions. A snowflake diagram captures the same class of schemas as star or snowflake schemas as defined by Kimball [18], but does it a the conceptual level.

The constraints on the snowflake diagram have been chosen with great care to ensure *summarizability* [20, 25, 28] of the data. Informally, summarizability means that the user gets the same (correct) result no matter if the aggregation of data to a more general level is performed in one or several steps. For summarizability to hold, two properties of the hierarchies must be satisfied. First, all cardinalities must be "-1" when going from the bottom of the dimension and up the hierarchy. This ensures that no data will be double-counted. Second, the minimum cardinality of the lower class in an aggregation relationship must be 1 rather than 0. This ensures that all data will be counted. Furthermore, the relationships from the fact class to the dimensions must be "-1" and go only to the lowest level in the dimension. This ensures that data will neither be double-counted or "get lost" when aggregating in several steps.

Example 4.1 UML Snowflake Diagram
As an example of a UML snowflake diagram consider Figure 2. The class "fact" is the root of the UML snowflake diagram. It has three dimensions, as indicated by the three associations between class "fact" and classes "ec", "customer", and "day". Each dimension is referred to as dimension "component", "customer", and "time", respectively. The "customer" dimension contains a single hierarchy, where classes "customer" and "country" each define a hierarchy level. The "time" dimension contains two hierarchies, where "day"-"week"-"year" and "day"-"month"-"quarter"-"year" are the hierarchy levels for the two hierarchies. This way, the time hierarchy links dates to either week-year or month-quarter-year, respectively. As an example, May 18th 2000 is linked to May 2000, which is linked to the second quarter of 2000, which is linked to the year 2000. The diagram has three *measures*, numbers that can be aggregated to higher levels, namely *sales_price*, *cost*, and *profit*.

We now define the sets used to describe a UML snowflake diagram.

SnowflakeDiagram = { (*class_name, AttributeList, PointsTo*) }

The set *SnowflakeDiagram* contains three-tuples where each tuple describes a class in the UML snowflake diagram. The first element in the tuple is a unique name identifying the class. The second and third tuple elements are both sets, where *AttributeList* is the set of attributes associated with a particular snowflake class and *PointsTo* is the set of classes for which links exist.

AttributeList = { (*att_name, att_type, modifier, AID, calc_att, link_path,type, aggfunc*) |
att_type $\in$ *DataTypes* $\wedge$ *modifier* $\in$ { ?, 1 } $\wedge$ *type* $\in$ {measure,descriptive}
$\wedge$ *aggfunc* $\in$ {SUM,COUNT,MIN,MAX,AVG,NULL} }

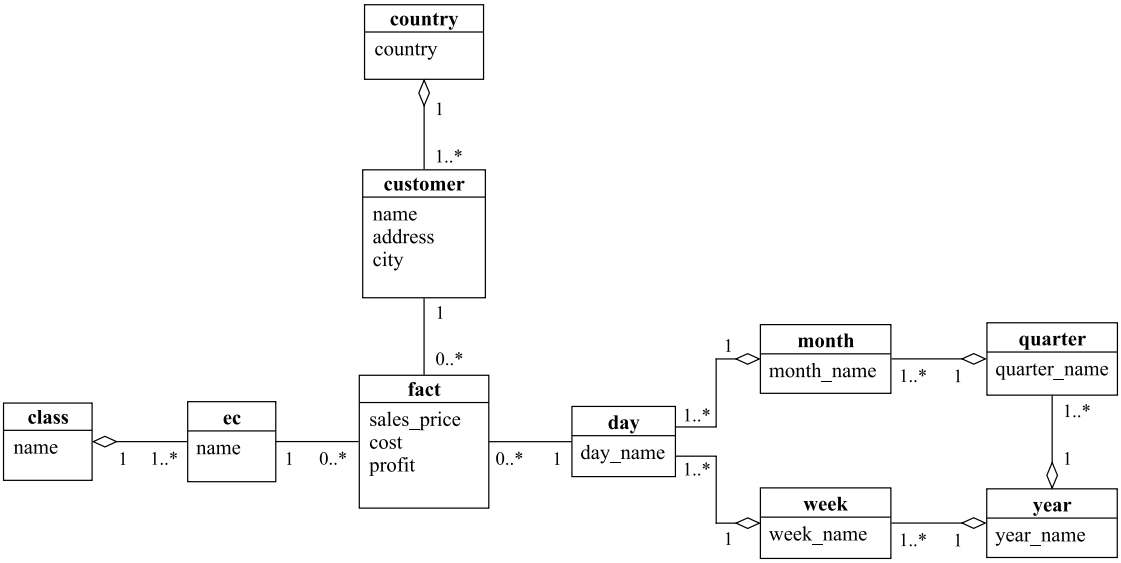

Figure 2: UML Snowflake Diagram

This set describes the name, data type, modifier, Attribute Identifier (AID), formula, link path, type, and default aggregate function for each attribute associated with a class. Here, *att_name* is a unique name identifying the attribute, *att_type* is the data type of the attribute, *modifier* indicates whether or not NULL values are allowed for the attribute, and *AID* is an identifier uniquely identifying the source and location of the attribute. The location of each attribute is described explicitly since all attributes in a class does not have to originate from the same data source. If the attribute originates from an XML document, the *AID* tuple element is a concatenation of the URI $r \in$ *UMLClasses*, the path in the XML document to the attribute, and the attribute's name in the XML document. If the attribute originates from a relational data source, the *AID* is a concatenation of the URI of the data source, the table name, and the attribute name. Further, *calc_att* is a formula used for calculating the attribute *att_name*. If a formula is associated with an attribute, the attribute is denoted a *calculated attribute*. All attributes which are not calculated attributes have the element *calc_att* set to NULL, indicating that this particular attribute is not dependent on any calculations. Furthermore, when defining a calculated attribute the *AID* tuple element is NULL, indicating that the calculated attribute do not exist in any data source. The *type* distinguish measure attributes from descriptive dimension attributes. For measures, the default aggregate function, which is used when aggregating the values to higher levels, has to be specified. For descriptive attributes, this is NULL.

The following context-free grammar (in Backus-Naur form) can be used for specifying calculated attributes, defining the expressions allowed for formulas. The grammar describes the construction of arithmetic expressions preserving the standard operator precedence. Parentheses are used to override this precedence. The grammar incorporates the possibility of using unary and binary functions (f and g, respectively) belonging to some function library (e.g. the standard functions normally used in relational DBMSs [21]).

$$\begin{aligned}
\langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{EXPR} \rangle - \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\
\langle \text{TERM} \rangle &\rightarrow \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{TERM} \rangle \div \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\
\langle \text{FACTOR} \rangle &\rightarrow (\langle \text{EXPR} \rangle) \mid \mathsf{f}(\langle \text{EXPR} \rangle) \mid \mathsf{g}(\langle \text{EXPR} \rangle, \langle \text{EXPR} \rangle) \mid a
\end{aligned}$$

The terminal symbol a is a metavariable defined as:

$a \in \mathbb{R} \cup \{ \textit{AttributeList}[\textit{att\_name}_j] \}$

The specification of the metavariable restricts calculated attribute formulas to only include real numbers and/or attributes from the class in which the calculated attribute is defined. *link_path* is a comma-separated list of *AID*s, attribute names, or *Link*s between classes listing in correct order the location of the attributes participating in the

establishment of the necessary relationships when adding attributes to a class. This element is set to NULL if the attribute is native within the class. The process of adding attributes is described below.

$Link = \{$ (*class_name, link_type, link_role*) $\}$

The *Link* set describes a link between two classes in a UML diagram, *class_name* is the name identifying the class for which a link exist, *link_type* is either an association or aggregation, and *link_role* is used for naming an association type link.

$$PointsTo = \{ (class\_name, link\_type, source\_card, target\_card) \mid source\_card \in \{ 0..*, 1..* \} \wedge target\_card = 1 \}$$

The *PointsTo* set describes links between classes in the UML snowflake diagram, *class_name* is the unique name identifying the class for which a link exist. Each link is a cardinality specified association or aggregation described by the attribute *link_type*. Further, *source_card* and *target_card* is the cardinality specified for the source and the target of the link, respectively. The value of *source_card* is depending on whether *link_type* is an association or aggregation, according to Definition 4.1. Finally, *target_card* is always set to 1.

Example 4.2 UML Snowflake Diagram Sets
Consider the UML snowflake diagram seen in Figure 2. The attribute list for dimension class "ec" is defined as:

$$AttributeList_{ec} = \{ (\text{ id, NUMERIC, 1, } URL_{component}\text{:class/ec.id, NULL, NULL }), (\text{ name, TEXT, 1, } URL_{component}\text{:class/ec.name, NULL, NULL }) \}$$

It specifies that class "ec" has two attributes; "id" and "name". Notice that the "id" attribute is implicit in the graphical representation of the UML snowflake diagram in terms of the association between class "ec" and class "fact". "name" has data type TEXT, and NULL values are not allowed for this attribute. Furthermore, "name" is located at $URL_{component}$:class/ec.name, where "class/ec.name" indicates that "name" is an attribute within the "ec" element nested beneath the "class" element. The two NULL-value elements in the tuple specifies that "name" is not a calculated attribute and that it has not been added to the class through a link path. The *PointsTo* set for class "fact" is defined as:

$$PointsTo_{fact} = \{ (\text{ ec, ASSOCIATION, 0..*, 1 }), (\text{ customer, ASSOCIATION, 0..*, 1 }), (\text{ day, ASSOCIATION, 0..*, 1 }) \}$$

It specifies that classes "ec", "customer", and "day" are the lowest hierarchy levels in the dimensions of "fact".

UML Snowflake Diagram Construction The designer constructs the UML snowflake diagram by choosing elements from various data sources describing the domain of the model. Each data source is described (visualized) by a UML diagram and the designer creates classes in the UML snowflake diagram by choosing one or more UML classes from diagrams describing the data sources. Several UML classes from the diagrams can be used to create one single UML snowflake diagram class, as well as one UML class in a diagram can be used to create multiple UML snowflake diagram classes. As mentioned above, this process is supported by built-in "design patterns" in the GUI design tool. Each class in the UML snowflake diagram is constructed by choosing one class in a UML diagram as the foundation for the UML snowflake diagram class. This class is extended by adding attributes from other classes, either from the same diagram or from UML classes originating from other data sources.

Example 4.3 Creating a Fact Class
Consider the UML diagram to the left in Figure 3 modeling sales of components. Each sale has a unique id identifying the sale ("salesID") and an id identifying the customer who bought the components ("customerID.value"). The class "item" describes the price and id of the sold components. The aggregation relationship between the two classes having cardinalities 1 and 1..* models that one or more components are sold to a customer at a time. The class "time" states the time of the sale.

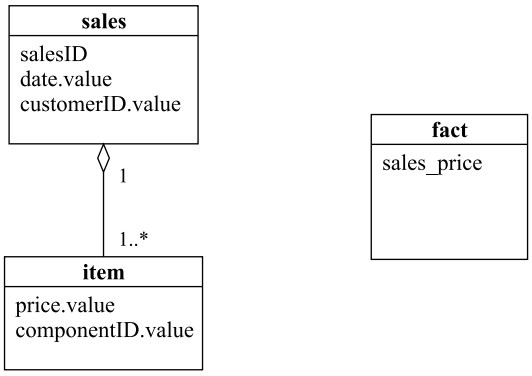

Figure 3: UML Diagrams Describing Sales and the Fact Class

A fact class for analyzing sales of components can be constructed from the classes in the UML diagram. In this case, the fact table is constructed from the classes "sales" and "item" which together contain the data to be used. The fact table is simply created by constructing a new element in the set *SnowflakeDiagram*, naming the class and setting the elements in the *AttributeList* to the name and location of the attributes selected to be used as fact data. The class "item" is used as the foundation for the fact class as it has the right cardinality (one occurrence per sale of a given component). Attributes from class "sales" is added to the fact class, yielding the following attribute list for the fact class (some of the attributes have been renamed):

$$
\begin{aligned}
\textit{AttributeList}_{\textit{fact}} = \{\ &(\ \text{customerID, NUMERIC, 1, } \textit{URL}_{\textit{sales}}\text{:sales.customerID.value,}\\
&\quad \text{NULL, NULL }),\\
&(\ \text{date, DATE, 1, } \textit{URL}_{\textit{sales}}\text{:sales.date.value, NULL, NULL }),\\
&(\ \text{internComponentID, NUMERIC, 1,}\\
&\quad \textit{URL}_{\textit{sales}}\text{:sales/item.componentID.value, NULL,}\\
&\quad (\ \text{(item, AGGREGATION, NULL)}\ )\ ),\\
&(\ \text{sales\_price, NUMERIC, 1, } \textit{URL}_{\textit{sales}}\text{:sales/item.price.value,}\\
&\quad \text{NULL, (\ (item, AGGREGATION, NULL)}\ )\ )\ \}
\end{aligned}
$$

The constructed fact class is illustrated to the right in Figure 3 containing the measure attribute "sales_price".

Adding Attributes to UML Snowflake Diagram Classes When constructing classes in the UML snowflake diagram each class often consists of a number of attributes originating from a single class in some data source. In many cases though, it is desirable to add additional attributes to the UML snowflake diagram class. As an example, if a retailer would like to analyze profit for specific products, data must exist that describe the actual sale of the product along with data stating the amount of money paid by the retailer for the product. If the UML class containing the attributes used to construct the fact class does not contain the required data, the attribute set for the class has to be extended with extra data. If a UML class T is extended with additional attributes from another UML class S_n, it is required that a path of UML classes $S_0, S_1, ..., S_n$ ($n \geq 0$) exists, where the cardinality of the relationship between S_0 and S_n is many-to-one. It is not required that the path from S_0 to S_n is located entirely within one UML diagram, making it possible to include attributes from different data sources. Furthermore, the attributes used to create new links between classes must have *modifier* equal to 1, in order to avoid NULL values in comparison expression. The following OQL expression is a general description of how to redefine class T to T', which includes in its definition the attributes from class T and the attributes $a_1, a_2, ..., a_p$ from a UML class S_n. Note that the OQL is only a tool for getting a precise specification, so the designer does not have to write OQL. Instead, the OQL is generated by the design GUI, allowing the designer to stay at the conceptual level.

SELECT $A \in T, S_{m+1}.S_{m+2}.\ ...\ .S_n.a_1,\ ...\ , S_{m+1}.S_{m+2}.\ ...\ .S_n.a_p$
FROM $T, S_0, S_1,\ ...\ , S_n$
WHERE $T.att_T = S_0.att_{S_0}$ AND
$S_0.S_1.\ ...\ .S_i.att_{S_i} = S_{i+1}.att_{S_{i+1}}$ AND
$S_{i+1}.S_{i+2}.\ ...\ .S_j.att_{S_j} = S_{j+1}.att_{S_{j+1}}$ AND
... AND
$S_k.S_{k+1}.\ ...\ .S_m.att_{S_m} = S_{m+1}.att_{S_{m+1}}$

The SELECT-clause lists the attributes in class T', in this case all the attributes A originally in T and the attributes a_1 to a_p ($p \geq 1$) from class S_n. The path expression $S_{m+1}.S_{m+2}.\ ...\ .S_n.a_p$ is used to retrieve the attribute a_p by going through the path $S_{m+1}.S_{m+2}.\ ...\ .S_n$, defined within one UML diagram.

Since it is not required that the path from S_0 to S_n is located entirely within one diagram, a connection between the diagrams used to retrieve the desired attributes is needed when adding attributes from another data source. This connection is established in the WHERE-clause, where the expression defines a path from T to a class S_{m+1}. The class S_{m+1} is located in the diagram in which the class containing the attributes $a_1, a_2, ..., a_p$ is located. In the WHERE-clause above, the attributes att_T and att_{S_0} are used to connect class T and S_0. The connection is established by matching on the values of these attributes. Each time a connection is established between two classes, an AND-clause is added to the expression. The reason for establishing an explicit connection in the WHERE-clause between the UML diagrams and not integrating the UML diagrams by adding associations or aggregations between classes, is that the diagrams may originate from different data sources and we want to eliminate the overhead involved in conceptually integrating the diagrams. Since the establishment of connections between classes in different data sources is based entirely on the designer's knowledge about the data sources, it is not possible to ensure that a many-to-one relationship exist between the involved classes, as is required to ensure summarizability. That this constraint really holds has to be checked at query time when the actual matching of attributes is performed. The FROM-clause states all the UML classes that are used in the attribute-adding process. The names of the added attributes $a_1, a_2, ..., a_p$ are defined by the designer. The only restriction is that the names of all attributes in class T' are distinct.

Example 4.4 Adding an Attribute
As an example of how to add an attribute to a class consider the following problem. A retailer would like to analyze profit of products sold. In order to calculate profit, data is needed that describe the cost of the product along with data stating the sales price of the product. As it can be seen from the UML diagram in Figure 4 (a) only data stating the sales price is present. Therefore, additional data is needed. The information needed (the purchase price) is available, but is located in another data source. The UML diagram of interest is diagram (c) in Figure 4, describing the components sold by the supplier. The component numbering system used by the retailer is different from the one used by the supplier, so in order to map between the two numbering systems, the UML diagram in Figure 4 (b) has been constructed by the retailer. As it can be seen from Figure 4, no explicit relationship exist between class "fact" (the class to be extended) and class "unitprice" (the class defining the desired attribute "price.value"). In order to retrieve the "price.value" attribute an explicit path has to be established from "fact" to "unitprice". This path is illustrated in Figure 4 as solid lines from diagram (a) through diagrams (b) and (c). The graphical illustration of path establishment and attribute retrieval (dotted line in Figure 4) is accomplished by executing the following OQL expression:

SELECT fact.internComponentID, fact.sales_price, ec.unitprice.price.value
FROM fact, product, ec, unitprice
WHERE fact.internComponentID = product.our_id AND product.id = ec.id

The retailer is able to extend the "fact" class with attribute "price.value" (renamed by the designer to "cost") because a many-to-one relationship exist between the attributes "internComponentID" and "our_id", and between the "id" attributes in classes "product" and "ec", respectively. These relationships are not "physically" present (in form of either associations or aggregations), but is tacit knowledge the retailer has about the domain. As described above,

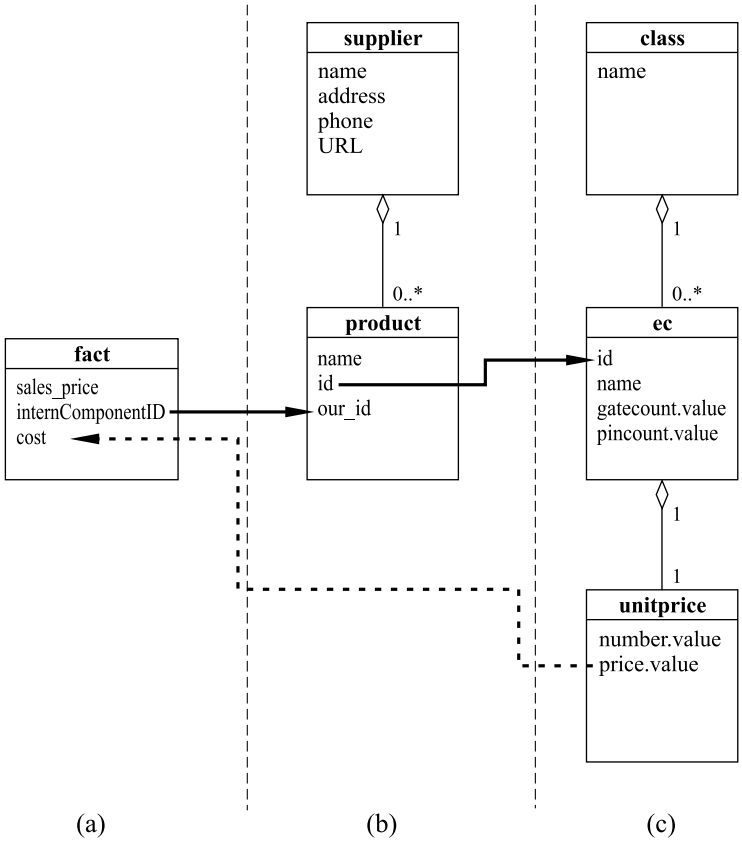

Figure 4: UML Diagrams (a), (b) and (c)

each attribute in a UML class is described by different types of elements. Illustrated next is the *AttributeList* element describing the attribute "cost" in class "fact". Notice the element *link_path* (the last element in the tuple) describing the ordered list of *AID*s locating the attributes used in the establishment of the necessary relationships. It is assumed that UML diagrams (a), (b), (c) in Figure 4 are located at URL_{fact}, $URL_{supplier}$ and $URL_{component}$, respectively.

(cost, NUMERIC, 1, $URL_{component}$:class/ec/unitprice.price.value, `NULL`,
(internComponentID, $URL_{supplier}$:supplier/product.our_id,
$URL_{supplier}$:supplier/product.id, $URL_{component}$:class/ec.id))

4.2 Specifying OLAP DBs on XML Documents

When specifying OLAP DBs with the UML snowflake diagram, some properties inherently connected to XML data can be exploited. As mentioned above, this is supported by built-in "design patterns" that the designer can choose from in order to ease the specification process.

The general structure of XML data is a hierarchical tree-structure of data elements caused by the nesting structure of XML. The nesting structure of XML is mostly used for grouping and subgrouping related information whereas the actual data is often found at the leaf level of the XML tree. [1, 27] This means that there is a high probability of finding relevant *fact* data for use in the OLAP DB at the leaf level of the XML tree. The fact data is the ultimate child in some nesting path originating from the root, corresponding to a hierarchical description of the fact data. In other words, for a given element in the XML tree, the element's ancestors together define a hierarchically organized dimension. The algorithm `generateUML` [16] is designed such that the hierarchical structure of the original XML document is preserved and visible in the UML diagram. In the generated UML diagram, the nesting relationship between elements in the XML document is preserved by relating the nested classes using aggregation. This corresponds closely to the "consists of" relationship implied by nesting elements in XML, and thereby also to

the multidimensional understanding of a hierarchy where the ALL-element (corresponding to the root in the XML document) defines the most general case [13]. However, a few special cases occurring in some XML designs needs special attention when specifying an OLAP DB using a UML diagram derived from a DTD. The special cases are:

- A UML class having multiple parents
- ID-references between elements
- Recursion between elements

The way of handling these special cases in the design of an OLAP DB is described below.

A UML class having more than one parent If a class has more than one parent in the UML diagram (i.e. it is pointed to by more than one class), it is ambiguous to which part of the overlying hierarchy the class belongs. This problem can be resolved in three ways. First, we can "split up" the overlying hierarchy in as many parts as the number of parents to the element, and then choose which part of the hierarchy to use. As an example of this, consider Figure 5. The class "unitprice" has two parents, "ec" and "device". If the purpose of the OLAP DB is to analyze the sales of components only (no devices are sold by the retailer), the designer - knowing that only components are sold - would choose to use only the part of the hierarchy where "ec" is parent of "unitprice".

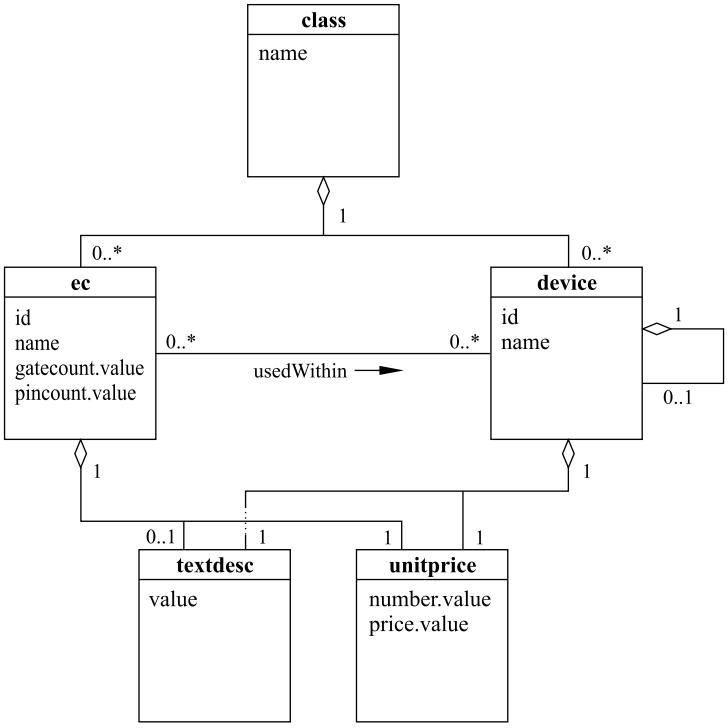

Figure 5: UML Diagram Derived From The Component Document DTD

Second, if the purpose of the OLAP DB is to analyse sales of both components and devices, but independently, the fact table and the product dimension has to be split by product type. This will result in two fact tables and two dimension tables, one set for components and one for devices. The splitting of heterogeneous fact tables and dimensions is a common multidimensional modeling issue and will not be explained in detail here, see [18] for a detailed explanation. Third, we can merge all the parents of a class into one single class. In the example described in the previous paragraph, this would mean that "ec" and "device" are merged into one class, acting as a single parent to "unitprice". This enables analysis of "unitprice" as a descendent of a new common class, which in this example could be called "item". Which solution to use is a decision to be made when designing the OLAP DB, depending on the purpose of the database.

ID-references between elements In XML it is possible to make references between elements using IDREF/ID constructs. These references are represented in the UML diagram as directed associations between the classes having an ID-reference relationship. When designing the OLAP DB, a decision has to be made of how to handle these references. In this case, two solutions are possible. The first solution is to simply ignore the reference. If making analysis of sales compared to classes of components, the designer could argue that the reference from "ec" to "device" (Figure 5) indicating that the component is used to build that particular device, is of no relevance to the analysis and therefore ignore the reference. The second solution is to include the information from the element being referred to in the referring element. In the example this would mean that the attributes from the "device" class are copied to the "ec" class, thereby capturing the relationship between "ec" and "device". To include information from the element being referred to, a many-to-one relationship must exist between the referring and the referred element in order to ensure summarizability. In case of an optional IDREF attribute or an IDREFS attribute this constraint is clearly not satisfied and therefore such data cannot be used in the specification.

Recursion between elements The possibility of defining elements recursively in XML causes some problems when designing the OLAP DB. Recursively defined elements can - in theory - generate an infinitely deep hierarchy so a decision has to be made about the maximum depth of the hierarchy. In the hierarchy generated by a recursive definition, the same class defines different levels in the hierarchy. As an example, in a recursively defined XML document having a recursive depth of n, the same class defines n different levels in the generated hierarchy. A hierarchy in OLAP terms is a hierarchy of generalization where the top level defines the most general case and the lowest level defines the most specific case. Again, two solutions are possible. First, the designer can choose to ignore the recursive definition by setting the maximum recursive depth to zero. Second, the recursive definition can be un-folded to a fixed depth, usually only 1 or 2, allowing the referenced information to be used in the specification of the cube. This choice must be made by the designer. However, the problem of recursion is not necessarily a big one. As claimed in [27], recursion, even though possible in XML, is not used extensively for conventional data modeling, making recursion unlikely to appear in the data used for building the OLAP DB.

4.3 Example: Construction of a UML Snowflake Diagram

The following example illustrates the construction of a UML snowflake diagram according to Definition 4.1. The UML snowflake diagram constructed is the diagram in Example 4.1 which enables analysis of component sales. The diagram has three dimensions describing customers, components, and time. The snowflake diagram is constructed from two different XML documents and one relational data source. The UML diagram derived from the *sales document* DTD, which describes sales of components is seen in Figure 3. It is assumed that the XML document is located at URL_{sales}. The *component document* describing the supplier's products used in this example is seen in Figure 5. It is assumed that the XML document is located at $URL_{component}$. Data describing *customers* is stored in a relational database. As assumed in Section 3 a UML diagram is available describing the relational data source. The URI stating the location of the data source is assumed to be $URI_{customer}$. For each customer, data is stored in the relational database stating the id of the customer, the name, the address (street and number), the city, and the country in which the customer lives.

We now proceed to describe the construction of a UML snowflake diagram from the data sources described above. The constructed snowflake diagram enables analysis of profit dimensioned by customer, component type, and time. The OLAP DB can be used to answer queries such as "What is the total profit of components belonging to class "semiconductor" sold in the year 2000". The intuition is to construct a fact class containing a measure stating the profit for each sale. Furthermore, dimensions describing customers, components, and time are added to the UML snowflake diagram. The following describe the construction of the fact class and the three dimensions. Due to space constraints, we only give textual descriptions here, the precise specifications can be seen in Appendix C.

The content of the *fact class* is primarily composed of data from the sales document (as described in Example 4.3). Additionally, to calculate the profit for a sale, data stating the cost of the product sold is needed. The attribute "cost" stating this amount is added from the component document as described in Example 4.4. The attribute "profit" is calculated by subtracting "cost" from "sales_price". The *component dimension* contains data originating only

from the component document. Only a subset of the document is used to create the dimension classes. The classes participating in the dimension are "ec" and "class", which due to their aggregation relationship can be used directly as a hierarchically organised dimension, thereby exploiting the hierarchically structure of the XML document. The data described by the UML diagram in Figure 6 is used to create the *customer dimension*. The dimension contains two hierarchy levels, "customer" and "country". "country" is chosen as a hierarchy level because it is more general than a specific address. The "customer" level contains the attributes "cust_id", "name", "address", and "city". The "country" level contains the attribute "country". The *time dimension* contains data originating only from the sales document. Only the "date" attribute in the "sales" class of the document is used to create the time dimension. As described in Example 4.1, the time dimension contains two hierarchies. The dimension has two hierarchies since we want to roll up from days to both weeks and months, but weeks do not roll up to neither months or quarters. The set *SnowflakeDiagram* is constructed by creating tuples for each class participating in the snowflake diagram, in this case "fact", "ec", "class", "customer", "country", "day", "week", "month", "quarter", and "year". The content of these tuples is the name, the *AttributeList*, and the *PointsTo* sets for each class. The resulting snowflake diagram is the seen in Figure 2. The diagram has three dimensions with two or three levels each and three measures, of which "profit" is calculated.

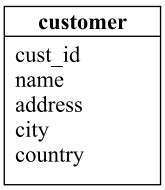

Figure 6: UML Diagram Describing Customer Data

5 Conclusion and Future Work

Motivated by the increasing use of OLAP tools for analyzing business data, and XML documents for exchanging information on the Web, this paper provides techniques that enable existing OLAP tools to exploit XML and relational data, without requiring physical integration of data.

This paper proposed a multidimensional model, the UML snowflake diagram, enabling a precise specification of an OLAP DB based on multiple XML and/or relational data sources. The UML diagramming method was used for describing and visualizing the logical structure of XML documents, easing the design of the OLAP DB. The paper described how to handle the special considerations that need to be taken when designing an OLAP DB on top of XML data. Also, an architecture for integrating XML data at the conceptual level was presented. The architecture also supported relational data sources, making it well suited for building OLAP DBs which are based partly on in-house relational data and partly on XML data available on the web. We improve on previous work on integration of web-based data by focusing on data integration at the conceptual rather than the logical level. Also, the data integration approach takes special OLAP issues, such as handling dimensions with hierarchies and ensuring summarizability, into account.

The implementation of a prototype using the approach described in this paper is currently in progress. A very important aspect of the implementation is to investigate efficient query processing techniques such as query translations and data caching. Storing higher-level summaries of the data can also speed up query processing considerably. Furthermore, if XML Schema advances to a W3C Recommendation it would be interesting to consider using this richer formalism for describing XML data sources instead of using DTDs. Other aspects of XML, such as whether preservation of document order is of relevance to OLAP analysis should also be investigated.

References

[1] Abiteboul, S. Querying Semistructured Data, *Proceedings of the Xth International Conference on Database Theory*, 1997.

[2] Abiteboul, S. et al. Tools for Data Translation and Integration. *IEEE Data Engineering Bulletin* 22(1), pp. 3-8, 1999.

[3] Bierman, G. M. *Using XML as an Object Interchange Format*, Department of Computer Science, University of Warwick, May 17, 2000, citeseer.nj.nec.com/374889.html. Current as of December 20th, 2000.

[4] Bonifati, A. et. al. Comparative Analyses of Five XML Query Languages. *SIGMOD Record* 29(1):68–79, 2000.

[5] Cattell, R. *The Object Database Standard: ODMG 3.0*, Morgan-Kaufmann, 2000.

[6] Chamberlin, D. et. al. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceedings of WebDB*, pp. 53–62, 2000.

[7] Computer Associates Corporation. ERwin Product Brochure. www.cai.com/products/alm/erwin/erwin_pd.pdf . Current as of January 18th, 2001.

[8] Deutsch, A. et al. Storing Semistructured Data with STORED. In Proceedings of *SIGMOD* , pp. 431–442, 1999.

[9] Fernandez, M. F. et al. Declarative Specification of Web Sites with Strudel. *VLDB Journal* 9(1), pp. 38–55, 2000.

[10] Florescu D. and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin* 22(3), pp. 27–34, 1999.

[11] E. Gamma et al. *Design Patterns.* Addison-Wesley, 1995.

[12] Garcia-Molina H. et al. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems* 8(2), pp. 117–132, 1997.

[13] Gray, J. et. al. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals, *Data Mining and Knowledge Discovery* 1(1):29–53, 1997.

[14] Hellerstein, J. M. et al. Independent, Open Enterprise Data Integration. *Data Engineering Bulletin*, 22(1):43–49, 1999.

[15] Hyperion Corporation. Hyperion Essbase OLAP 6. www.hyperion.com/essbaseolap.cfm. Current as of January 9th, 2001

[16] Jensen, M. R., T. H. Møller, and T. B. Pedersen. Converting XML Data to UML Diagrams For Conceptual Data Integration. *Submitted for publication*.

[17] Kimball, R. et. al. *The Data Warehouse Lifecycle Toolkit*, Wiley, 1998.

[18] Kimball, R. *The Data Warehouse Toolkit*, Wiley, 1996.

[19] Lahiri, T. et. al. Ozone: Integrating Structured and Semistructured Data. In *Proceedings of the Seventh International Conference on Database Programming Languages*, 1999.

[20] Lenz, H. et. al. Summarizability in OLAP and Statistical Databases, *Proceedings of the Ninth International Conference on Statistical and Scientific Database Management*, pp. 39-48, 1997.

[21] Melton, J. et. al. *Understanding the New SQL: A Complete Guide*, Morgan-Kaufmann, 1995

[22] Microsoft Corporation. Microsoft SQL Server 2000 Analysis Services White Paper. www.microsoft.com/sql/ productinfo/analysisservicesWP.htm. Current as of January 9th, 2001.

[23] Object Management Group. *OMG Unified Modeling Language Specification 1.3*. www.rational.com/uml/ resources/documentation/index.jsp, 1999. Current as of December 13th, 2000.

[24] Oracle Corporation. Oracle Express OLAP. www.oracle.com/ip/analyze/warehouse/bus_intell/index.html. Current as of January 9th, 2001.

[25] Pedersen, Torben Bach et. al. Extending Practical Pre-Aggregation in On-Line Analytical Processing, *Proceedings of the Twenty-Fifth International Conference on Very Large Databases*, pp. 663-674, 1999

[26] Pedersen, T. B. et. al. Extending OLAP Querying To External Object Databases. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, pp. 405–413, 2000.

[27] Pinnock, J. et. al. *Professional XML*, Wrox Press, 2000.

[28] Rafanelli, M. et. al. STORM: A Statistical Object Representation Model. In *Proceedings of the Fifth Conference on Statistical and Scientific Database Management*, pp. 14–2.59, 1990.

[29] Roth, M. T. et al. The Garlic Project. In Proceedings of *SIGMOD* , pp. 557, 1996.

[30] Shanmugasundaram, J. et. al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the Twenty-Fifth International Conference on Very Large Databases*, 1999.

[31] Silicon Integration Initiative. *The Electronic Component Information Exchange QuickData Architecture*, www.si2-.org/ecix/. Current as of December 29th, 2000

[32] Thomsen, E. et. al. *Microsoft OLAP Solutions*, Wiley, 1999.

[33] Thomsen, E. *OLAP Solutions: Building Multidimensional Information Systems*, Wiley, 1997.

[34] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation. www.w3.org/TR/2000/REC-xml-20001006. Current as of December 20th, 2000.

[35] World Wide Web Consortium *XML Schema*, W3C Candidate Recommendation, www.w3.org/XML/Schema.html. Current as of December 30th, 2000.

A Detailed Example

This section provides the details of the example in Section 2.

First, the DTD for the mapping document is shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<!ELEMENT supplierDB (supplier*) >
<!ELEMENT supplier (product*) >
<!ATTLIST supplier name CDATA #REQUIRED
                   address CDATA #REQUIRED
                   phone CDATA #IMPLIED
                   URL CDATA #IMPLIED >
<!ELEMENT product EMPTY >
<!ATTLIST product name CDATA #REQUIRED
                  id CDATA #REQUIRED
                  our_id CDATA #REQUIRED >
```

An example of an XML "mapping" document is shown below. The document describes a mapping between the retailer's component id r12200 (a 2K2 carbon resistor) and the supplier "Component Heaven"'s component id 263854.

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE supplierDB SYSTEM "mapping.dtd">
<supplierDB>
  <supplier name="ComponentHeaven"
            address="457 Madison Ave., NY"
            phone="212-753-1722"
            URL="http://www.componentheaven.com/products">
    <product name="2K2 carbon resistor"
             id="263854" our_id="r12200"/>
    ...
  </supplier>
  ...
</supplierDB>
```

Next, we show the DTD for the component document.

```
<?xml version="1.0" encoding="utf-8" ?>
<!ELEMENT products (class+) >
<!ELEMENT class ((ec*, device*)) >
<!ATTLIST class name CDATA #REQUIRED >
<!ELEMENT ec (unitprice, pincount, gatecount, textdesc?) >
<!ATTLIST ec id ID #REQUIRED usedWithin IDREFS #IMPLIED
             name CDATA #REQUIRED >
<!ELEMENT device (textdesc, unitprice, device?) >
<!ATTLIST device id ID #REQUIRED name CDATA #REQUIRED >
<!ELEMENT unitprice (number, price) >
<!ELEMENT pincount (#PCDATA) >
<!ELEMENT gatecount (#PCDATA) >
<!ELEMENT textdesc (#PCDATA) >
<!ELEMENT number (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```

An example of an XML document conforming to component DTD is shown below. The document describes the two components "2K2 carbon resistor" and "8bit ALU". The resistor belongs to class "resistor", has 2 pins and no gates, and is sold in boxes of 250 for $6.25. The ALU belongs to class "semiconductor", has 28 pins, 192 gates and is sold in boxes of 10 for $8.70.

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE products SYSTEM "products.dtd">
<products>
  <class name="resistor">
    <ec id="r12200" name="2K2 carbon resistor">
      <unitprice>
        <number> 250 </number>
        <price> 6.25 </price>
      </unitprice>
      <pincount> 2 </pincount>
      <gatecount> 0 </gatecount>
    </ec>
    ...
  </class>
  <class name="semiconductor">
    <ec id="alu8003" name="8bit ALU">
      <unitprice>
        <number> 10 </number>
        <price> 8.70 </price>
      </unitprice>
      <pincount> 28 </pincount>
      <gatecount> 192 </gatecount>
      <textdesc> 8 bit Arithmetic Logical Unit </textdesc>
    </ec>
    ...
  </class>
  ...
</products>
```

B UML Query Language

This section explains precisely how OQL kan be used as a qeury language for UML diagrams.

One of the reasons that a subset of UML has been chosen as the conceptual modeling language used to describe XML and relational data sources, is the similarity between UML and the ODMG (Object Data Management Group) model [5]. The similarity between the subset of UML used in this paper and the ODMG model enables the use of Object Query Language (OQL) as a language for querying UML directly, since a complete mapping exists between the subset of UML used and the ODMG model. To explicitly state the similarity between a UML class and a class in the ODMG model, Object Definition Language (ODL) declarations are given for an example UML diagram below.

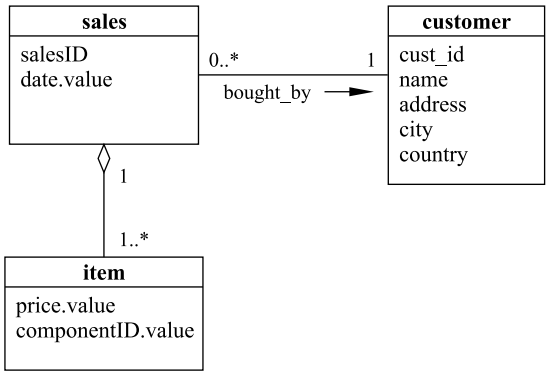

Figure 7: UML diagram

The UML diagram in Figure 7 represents all of the UML modeling constructs used for conceptual modeling in this paper, which are classes, typed attributes, and cardinality and role specified associations and aggregations. The attributes in the UML diagram depicted in Figure 7 all have their *modifier* set to 1, indicating that null-values are not allowed. The data type of the attributes "salesID", "price.value", "componentID.value", and "cust_id" is NUMERIC, "date.value" is of type DATE, and the remaining attributes are of type TEXT.

ODL Declaration for UML Class "sales"

```
interface sales {
   attribute numeric salesID not null;
   attribute date date.value not null;
   relationship Set<item> item_relation
                inverse item::sales_relation not null;
   relationship customer bought_by not null;
};
```

ODL Declaration for UML Class "item"

```
interface item {
   attribute numeric price.value not null;
   attribute numeric componentID not null;
   relationship sales sales_relation
                inverse sales::item_relation not null;
};
```

ODL Declaration for UML Class "customer"

```
interface item {
   attribute numeric cust_id not null;
   attribute text name not null;
   attribute text address not null;
   attribute text city not null;
   attribute text country not null;
};
```

These three ODL declarations represents the UML diagram in Figure 7. The `not null` constraint applied to each of the attributes models that all attributes in the UML diagram have their *modifier* set to 1.

The declaration of a `relationship` in classes "sales" and "item" models the relationship between these two UML classes. Notice that the two relationship-declarations are different, where relationship `item_relation` has type `Set<item>` and relationship `sales_relation` has type `sales`. These two declarations models the specified cardinalities in the UML diagram, where the declaration of a set declared `not null` models a 1..* relationship, whereas a relationship not defined as being a set models either 0..1 or 1 depending on whether or not the `not null` constraint is applied. [3]

The relationship `bought_by` is only declared in class "sales", indicating that the relationship exist only in the direction from "sales" to "customer". Uni-directional relationships in ODL models directed associations in UML.

The relationship attributes `item_relation`, `bought_by`, and `sales_relation` are implicitly present in the UML diagram for each association and aggregation type link.

C Detailed Snowflake Diagram Example

This section contains the precise specification of the snowflake diagram in Section 4.3.

Fact Class The set $AttributeList_{fact}$ is shown below.

$AttributeList_{fact} = \{$ (customerID, NUMERIC, 1, URL_{sales}:sales.customerID.value, NULL, NULL),
(date, DATE, 1, URL_{sales}:sales.date.value, NULL, NULL),
(internComponentID, NUMERIC, 1, URL_{sales}:sales/item.componentID.value, NULL, ((item, AGGREGATION, NULL))),
(sales_price, NUMERIC, 1, URL_{sales}:sales/item.price.value, NULL, ((item, AGGREGATION, NULL))),
(componentID, NUMERIC, 1, $URL_{supplier}$:supplier/product.id, NULL, (internComponentID, $URL_{supplier}$:supplier/product.our_id))
(cost, NUMERIC, 1, $URL_{component}$:class/ec/unitprice.price.value, NULL, (componentID, $URL_{component}$:class/ec.id)),
(profit, NUMERIC, 1, NULL, sales_price - cost, NULL) $\}$

$PointsTo_{fact}$ is the set of classes that "fact" is connected to through associations with cardinalities as defined in Definition 4.1. The classes participating in the *PointsTo* set is the classes "ec", "customer", and "day", indicating that these are the lowest hierarchy levels in the dimensions to the fact class.

$PointsTo_{fact} = \{$ (ec, ASSOCIATION, 0..*, 1), (customer, ASSOCIATION, 0..*, 1),
(day, ASSOCIATION, 0..*, 1) $\}$

Component Dimension The set *AttributeList* for the dimension class "ec" is shown below.

$AttributeList_{ec} = \{$ (id, NUMERIC, 1, $URL_{component}$:class/ec.id, NULL, NULL),
(name, TEXT, 1, $URL_{component}$:class/ec.name, NULL, NULL) $\}$

The *PointsTo* set for class "ec" is shown below. The set contains an aggregation type link, indicating that class "class" is a hierarchy level immidiately above the "ec" level.

$PointsTo_{ec} = \{$ (class, AGGREGATION, 1..*, 1) $\}$

The set *AttributeList* for the dimension class "class" is shown below.

$AttributeList_{class} = \{$ (name, TEXT, 1, $URL_{component}$:class.name, NULL, NULL) $\}$

The *PointsTo* set for class "class" is shown below. The set is empty, indicating that "class" is the top level in the component dimension hierarchy.

$PointsTo_{class} = \{\ \}$

Customer Dimension The set *AttributeList* for the dimension class "customer" is shown below. The Attribute Identifier (AID) for the attributes is the URI stating the location of the relational source, the table name, and the name of the attribute.

$AttributeList_{customer} = \{$ (cust_id, NUMERIC, 1, $URI_{customer}$:customer.cust_id, NULL, NULL),
(name, TEXT, 1, $URI_{customer}$:customer.name, NULL, NULL),
(address, TEXT, 1, $URI_{customer}$:customer.address, NULL, NULL),
(city, TEXT, 1, $URI_{customer}$:customer.city, NULL, NULL) $\}$

The *PointsTo* set for class "customer" is shown below. The set contains an aggregation type link, indicating that class "country" is a hierarchy level immediately above the "customer" level.

$PointsTo_{customer}$ = { (country, AGGREGATION, 1..*, 1) }

The set *AttributeList* for the dimension class "country" is shown below.

$AttributeList_{country}$ = { (country, TEXT, 1, $URI_{customer}$:customer.country, NULL, NULL) }

The *PointsTo* set for class "country" is shown below. The set is empty, indicating that "country" is the top level in the customer dimension hierarchy.

$PointsTo_{country}$ = { }

Time Dimension The sets *AttributeList* and *PointsTo* for each class in the time dimension are shown below.

$AttributeList_{day}$ = { (day_name, TEXT, 1, URL_{sales}:sales.date, NULL, NULL)
(date, DATE, 1, URL_{sales}:sales.date, NULL, NULL)
(week_id, NUMERIC, 1, URL_{sales}:sales.date, NULL, NULL)
(month_id, NUMERIC, 1, URL_{sales}:sales.date, NULL, NULL) }
$PointsTo_{day}$ = { (week, AGGREGATION, 1..*, 1), (month, AGGREGATION, 1..*, 1) }

$AttributeList_{week}$ = { (week_name, TEXT, 1, URL_{sales}:sales.date, NULL, NULL)
(week_id, NUMERIC, 1, URL_{sales}:sales.date, NULL, NULL)
(year_id, NUMERIC, 1, URL_{sales}:sales.date, NULL, NULL) }
$PointsTo_{week}$ = { (year, AGGREGATION, 1..*, 1) }

$AttributeList_{month}$ = { (month_name, TEXT, 1, URL_{sales}:sales.date, NULL, NULL)
(month_id, NUMERIC, 1, URL_{sales}:sales.date, NULL, NULL)
(quarter_id, NUMERIC, 1, URL_{sales}:sales.date, NULL, NULL) }
$PointsTo_{month}$ = { (quarter, AGGREGATION, 1..*, 1) }

$AttributeList_{quarter}$ = { (quarter_name, TEXT, 1, URL_{sales}:sales.date, NULL, NULL)
(quarter_id, NUMERIC, 1, URL_{sales}:sales.date, NULL, NULL)
(year_id, NUMERIC, 1, URL_{sales}:sales.date, NULL, NULL) }
$PointsTo_{quarter}$ = { (year, AGGREGATION, 1..*, 1) }

$AttributeList_{year}$ = { (year_name, TEXT, 1, URL_{sales}:sales.date, NULL, NULL)
(year_id, TEXT, 1, URL_{sales}:sales.date, NULL, NULL) }

The *PointsTo* set for class "year" is shown below. The set is empty, indicating that "year" is the top level in the time dimension hierarchies.

$PointsTo_{year}$ = { }

Snowflake Diagram The precise contents of the snowflake diagram is specified below.

SnowflakeDiagram = { (fact, $AttributeList_{fact}$, $PointsTo_{fact}$),
(ec, $AttributeList_{ec}$, $PointsTo_{ec}$),
(class, $AttributeList_{class}$, $PointsTo_{class}$),
(customer, $AttributeList_{customer}$, $PointsTo_{customer}$),
(country, $AttributeList_{country}$, $PointsTo_{country}$),
(day, $AttributeList_{day}$, $PointsTo_{day}$),
(week, $AttributeList_{week}$, $PointsTo_{week}$),
(month, $AttributeList_{month}$, $PointsTo_{month}$),
(quarter, $AttributeList_{quarter}$, $PointsTo_{quarter}$),
(year, $AttributeList_{year}$, $PointsTo_{year}$) }