Constructing OLAP Cubes from XML Data

Mikael Rune Jensen *Thomas Holmgren Møller*
mrj@cs.auc.dk *thm@cs.auc.dk*

Institute of Computer Science
Aalborg University

12th January 2001

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. In the traditional data warehousing approach, selected information is extracted in advance and stored in a repository. This approach is widely adopted because of its high-performance guarantee. However, in some situations a logical integration of data is preferable since some data is inherently not suited for storing in a data warehouse, e.g. dynamically evolving information.

Extensible Markup Language (XML) is fast becoming the new standard for data representation and exchange on the World Wide Web. The rapid emergence of business-to-business (B2B) e-commerce on the Web based on XML documents is making analysis tools capable of handling XML data necessary.

Based on a real-world case study this paper presents techniques for constructing an OLAP database (OLAP DB) from multiple data sources, enabling existing OLAP tools to exploit XML data.

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 [13, 21] and visual querying [27] supported by OLAP tools ease the process of decision support compared to traditional database management systems. Furthermore, OLAP tools offer better performance for aggregate queries by using pre-aggregation of data [19] and special multidimensional index structures.

The traditional 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 special kinds of data (e.g. dynamically evolving information). Enabling use of these kinds of data requires a logical integration where data is fetched on-demand, avoiding a 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 publicly available on the Web makes it attractive to use in conjunction with OLAP tools.

The contribution of this paper is to enable OLAP analysis directly on XML data using existing OLAP tools. This is achieved by creating a logical integration of heterogeneous data sources, enabling analysis of internal data in conjunction with external data, avoiding problems associated with physical integration of data. Furthermore, to ease the integration of data, methods for enabling fast and easy browsing of XML data sources are presented.

To the best of the author's knowledge, this paper is the first to address the issue of logically integrating XML and relational data for use by OLAP tools. We also believe we are the first to provide methods for enabling easy and fast browsing of XML data.

Related work exist in areas such as extension of OLAP queries to external databases [18], integration of structured and semistructured data [12], storing XML data in relational databases [22], and the issue of querying XML data [3, 5].

The work covered in [18] presents a federated system extending OLAP queries to object databases. A similar federated approach could be taken for extending OLAP queries to XML data, but following the work in [18], this would require the invention of a new query language and a new OLAP system. In comparison, our approach utilizes existing OLAP tools and hence requires no new query language.

The "Ozone" system [12] integrates structured and semistructured data. The Ozone system proposes a physical integration of structured and semistructured data, where the main goals are to store data efficiently while retaining the structure (or lack of structure) of the semistructured data. In comparison, our goal is to create a logical integration of XML and relational data adding multidimensional structure to the data suitable for OLAP analysis.

The issue of converting XML data into relational data is described in [22]. The main contribution is to provide a method for storing XML data efficiently in relational databases, while retaining the structure of the XML data. The methods presented in [22] cannot be used directly, since the structure of the generated relational data is not suitable for OLAP queries, and because the proposed methods aims at building a physical database rather than a logical integration.

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 used for OLAP analysis. Section 4 defines algorithms for generating a UML diagram from an XML DTD. Section 5 defines a multidimensional model capturing the structure and content of a logical integration of XML and relational data sources constituting an OLAP DB. Section 6 discusses issues associated with populating data structures generated from the multidimensional model. As a foundation for data retrieval, a discussion concerning different semistructured query languages is also presented. Section 7 summarizes and points to topics for future research.

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 model, 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) [24]. 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 [24].

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 [28, 20].

2.1 Information Stored by the Retailer

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.

2.1.1 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>
```

2.1.2 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 is shown below. Every mapping document conforms to this DTD.

```
<?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 document conforming to the above DTD 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>
```

2.1.3 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_relation(id : integer,
                  name : text,
                  street : text,
                  number : text,
                  city : text,
                  country : text)
```

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

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

2.2 Information Provided by the Supplier

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 is shown below. Every component document conforms to this DTD.

```
<?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 the above 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>
```

Based on the above data, it is the goal of the retailer to construct an OLAP DB capable of answering questions such as “What is the net profit of sales of semiconductors this year?”, “What is the best selling classes of components?”, and “In which country do we sell the most resistors?”. This paper describes the construction of an OLAP DB capable of answering these questions. The OLAP DB is based on a mixture of the data stored locally by the retailer and the data provided as XML on the Web by the supplier, in order to answer the above questions. Elements from the case study is used as examples throughout this paper.

3 Architectural Overview

This section describes a general system architecture for logically integrating XML and relational data sources for use by OLAP tools.

The architecture makes the use of XML data transparent to the OLAP tool and thereby to the end user, avoiding the problems inherently connected to inventing (and learning) a new query language.

3.1 XML Data Sources

Before the designer can construct an OLAP DB 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. What is needed is an easy understandable and fast 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 is 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 standardized, graphical modeling language, each XML data source on the Web becomes an easy accessible and comprehensible database. 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. Section 4 gives a detailed description of the transformation from DTDs to UML diagrams.

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 [31], but the DTD is the only formalism that is a World Wide Web Consortium (W3C) Recommendation. The XML 1.0 Recommendation [28] is used throughout this paper.

3.2 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 way of visualizing 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.

3.3 General System Architecture

The two types of data sources are integrated by “disguising” the XML data as relational data and creating a relational multidimensional view (a star or snowflake schema [11]) over the data sources.

Figure 1 illustrates the architecture which enables the use of XML data and/or relational data for performing decision support using existing OLAP tools.

The central entity in the architecture is the Data Integrator which enables XML data and/or relational data to be used by an OLAP tool. When XML data is used it is “disguised” as a relational multidimensional schema which are combined with the relational data sources and made available to the OLAP tool 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.

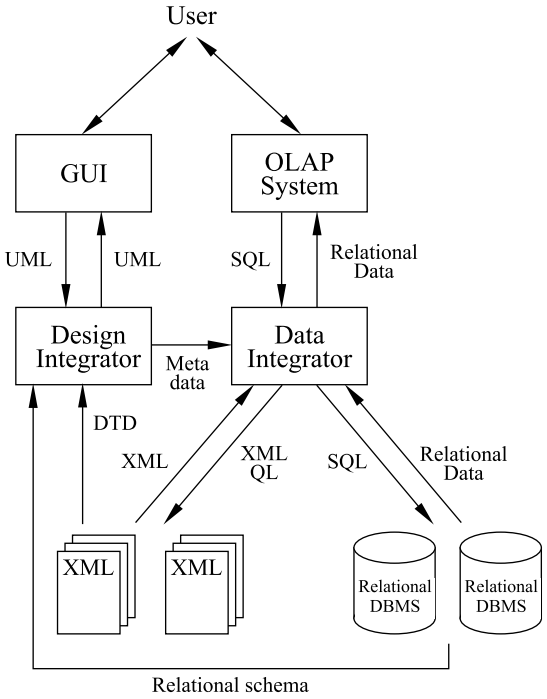

Figure 1: Architecture

An intermediate step in the construction of relational structures is the construction of a multidimensional model, which is transformed into a relational schema describing an OLAP DB.

The multidimensional model (termed a UML snowflake diagram) is constructed by the designer through a graphical user interface (GUI). The UML snowflake diagram is constructed from UML diagrams describing the data sources, which are generated by the Design Integrator. The construction of a UML snowflake diagram is described in detail in Section 5. The UML snowflake diagram is transformed into relational structures by the Data Integrator, and are then made available to the OLAP System.

The relational OLAP (ROLAP) approach for enabling OLAP analysis on XML data is used since ROLAP, as opposed to multidimensional OLAP (MOLAP), is a more general approach for enabling multidimensional analysis, since ROLAP is supported by both ROLAP and MOLAP tools [9, 17, 26] (through a multidimensional logical view over the relational structures) as well as standard non-OLAP query tools.

At query time, the 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 [15]. 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.

4 Easy Browsing of XML Data

This section describes algorithms for enabling fast and easy browsing of XML data available on the Web. The structure of XML data is visualized by a UML diagram derived from a DTD describing the XML data source. Furthermore, suggestions for determining types for XML elements are briefly presented.

The purpose of the algorithms is to build a UML diagram which is at least as "big" as the document described by a DTD, meaning that any XML document conforming to the DTD must be expressible in terms of the derived UML diagram.

The algorithms generate UML according to the OMG Unified Modeling Language Specification version 1.3 [16]. Only a subset of the modeling constructs described in this specification are used, namely classes, typed attributes, and cardinality and role specified associations and aggregations. The algorithms are designed in a way such that the structure of the original XML document is preserved and visible in the UML diagram.

The UML diagram has been chosen as the way of modeling and visualizing the DTD because UML is a standardized data modeling language which is familiar to many users, and because UML is powerful enough to express any document described by a DTD. The diagram is a conceptual easy understandable way of visualizing the structure of the XML document and allows the user quickly to get a feeling of the content and the structure of the data available. This supports easy browsing of data made available through XML on the web.

The algorithms for deriving a UML diagram from a DTD can in addition to the nesting constructs of XML also handle recursion between elements (both direct and indirect recursion by either nesting or reference) as well as references between different elements (ID/IDREF(S) constructs).

4.1 Assumptions

It is assumed that all ID-references (IDREFS referring to some IDs) from any element is a reference to one element type only. This means that for an element having multiple references defined (using an IDREFS attribute), each reference must be to the same element type. This assumption has been made since it in common data models (like the UML model, E-R model, and relational model) not is possible to model references to multiple unspecified elements, otherwise than making zero-to-many relationships between all entities in the model, a rather unsatisfactory solution. It is therefore assumed that the data contained within the XML documents is "well behaved" in the sense that all references are to one specific element type, even though arbitrary references are allowed in XML.

It is furthermore assumed that it is possible to determine the relationship between any two elements having an ID/IDREF(S) relationship. This means that for every element having an IDREF or IDREFS attribute defined, it is possible to determine the element type to which a reference exist.

4.2 Pre-processing of the DTD

Before generating the UML diagram corresponding to the document described by the DTD, the DTD is simplified in order to make further processing easier. This simplifying process is here termed DTD pre-processing.

It is not the intention to create a DTD that is equivalent to the original DTD. Instead the transformations creates a DTD which can be "less strict" than the original DTD, meaning that any XML document described by the original DTD is "at least" described by the transformed DTD. This assures that any document conforming to the original DTD also conforms to the transformed DTD.

All that matters in an XML document is the position of an element relative to its siblings and the parent-child relationship between elements, and it is therefore safe to apply the transformations since they preserve these conditions.

The pre-processing makes the following changes to the DTD:

- Removes elements not accessible from the document root
- Removes elements declared EMPTY which have not declared an ATTLIST
- Simplifies element content specifications

The first step towards simplifying the DTD is to remove all elements not accessible from the root. It is safe to apply this procedure since any XML document conforming to the DTD cannot make use of this type of elements. [28]

All elements declared EMPTY which does not have an ATTLIST declared can safely be removed from the DTD. It is possible to make use of this type of elements in an XML document, but the element can contain no data and are therefore of no interest. [28]

Most of the complexity of a DTD stems from the complex definition of element content specifications. By applying certain transformations to a DTD it is possible to simplify the DTD element content specifications. The transformations used in the pre-processing are a slight change of the transformations presented in [22], consisting in that "+" operators are not transformed into "*" operators. This is not done since it is possible to capture the "+" operator semantics in the UML model. Only a subset of the transformations are shown here.

Flattening transformations

$(e_1,\ e_2)^* \rightarrow e_1^*, e_2^*$
$(e_1,\ e_2)? \rightarrow e_1?, e_2?$
$(e_1 \mid e_2) \rightarrow e_1?, e_2?$

Simplification transformations

$e_1^{**} \rightarrow e_1^*$
$e_1^*? \rightarrow e_1^*$
$e_1?^* \rightarrow e_1^*$
$e_1?? \rightarrow e_1?$

Grouping transformations

$..., a^*, ..., a^*, ... \rightarrow a^*, ...$
$..., a^*, ..., a?, ... \rightarrow a^*, ...$
$..., a?, ..., a^*, ... \rightarrow a^*, ...$
$..., a?, ..., a?, ... \rightarrow a^*, ...$
$..., a, ..., a, ... \rightarrow a^+, ...$

The first set of transformations convert a nested definition into a flat representation, meaning that the binary operators "," and "|" do not appear inside any operator. The second set of transformations reduces many unary operators to a single unary operator. The last set of transformations group sub-elements having the same name.

Example 4.1 Simplification of Element Content Specification
As an example of how to simplify an element content specification consider the specification $<$*!ELEMENT* x $((b|c)^*, a, (d?|(e?, (b, b)^*))^*, a)$ $>$. By applying the transformations it can be reduced to $<$*!ELEMENT* x $(b^*, c^*, a^+, d^*, e^*)$ $>$.

4.3 DTD and UML Data Model

This section defines data models used to describe the DTD and the UML diagram components. Both the DTD and the UML data model are defined by means of an environment based on sets. The two data models are defined in order to give a precise description of the generation and modification of a UML diagram.

4.3.1 Notation

In order to ease the readability and understandability of the algorithms presented, the following notation will be used.

All sets are written in *italics* with initial capital letter. A capital letter is used to separate words of a set, e.g. *MySet*.

All elements of sets are written in *italics* with small letters. An underscore character is used to separate words of an element, e.g. *my_element*.

All functions are written in sans serif with small letters. A capital letter is used to separate words of a function, e.g. myFunction.

When a set consists of elements that are tuples square brackets ("[" and "]") are used to address a certain tuple element, e.g. when an element is defined by *my_element* $=$ $(my\_tuple\_element1, MyTupleElement2)$ then *my_element*$[my\_tuple\_element1]$ denotes the first element in the tuple. Furthermore, when a tuple consists of elements which are sets (like the second tuple from the former example) then a subscript will denote a certain element within this set, e.g. *my_element*$[MyTupleElement2_j]$ denotes element j in the set *MyTupleElement*2.

4.3.2 Sets Describing the Content of the Pre-processed DTD

This section defines the sets needed in order to describe an XML DTD.

ElementNames = $\{ e \mid e$ is an element name defined in the DTD $\} \cup$
$\{$ CDATA, PCDATA $\}$

The set *ElementNames* contains all names of elements declared with *!ELEMENT* in the DTD including the pre-defined character data elements CDATA and PCDATA.

AttributeNames = { a | a is an attribute name defined in the DTD }

AttributeNames contains all names of attributes declared with *!ATTLIST* in the DTD.

DTDContent = { (*element_name, ElementContentSpec, AttContentSpec*) |
element_name ∈ *ElementNames* }

The *DTDContent* set is a set of three-tuples describing the contents and structure of the DTD. The first element in the tuple is the name of an element from the DTD. The second and third elements in the tuple are both sets which are defined below.

ElementContentSpec = { (*element_name, modifier*) | *element_name* ∈ *ElementNames* ∧
modifier ∈ *Modifiers* \ { REQUIRED, IMPLIED } }

This set describes the content specification for all elements defined in the DTD. The set contains a two-tuple for each element listed in the element content specification for some element in the DTD. A modifier is associated with each element in accordance to the modifier specified in the DTD.

AttContentSpec = { (*att_name, att_type, modifier*) | *att_name* ∈ *AttributeNames* ∧
att_type ∈ *AttributeTypes* ∧ *modifier* ∈ *Modifiers* \ { *, +, ?, 1 } }

For each element declared in the DTD a list of attributes can be associated with the element. The set *AttContentSpec* contains three-tuples where the first element in the tuple is the name of the declared attribute, the second element is the type of the attribute and the third element is the modifier associated with the attribute.

Modifiers = { *, +, ?, 1, REQUIRED, IMPLIED }

Modifiers is the set of modifiers used to describe the cardinality of elements and attributes specified in the DTD.

AttributeTypes = { ID, IDREF, IDREFS, CDATA }

All attributes defined in a DTD are typed. *AttributeTypes* is the set of attribute types.

4.3.3 Definitions

Definition 4.1 Leaf Element
A leaf element is an element declared in the DTD having a content specification of some character data type (e.g. CDATA or PCDATA).

4.3.4 Sets Describing the UML Diagram

This section defines the sets for describing the UML diagram.

UMLClasses = { (*class_name, AttributeList, PointsTo*) | *class_name* ∈ *ElementNames* } ∪
{ r | r is a Uniform Resource Identifier }

The set *UMLClasses* describes a complete UML diagram. It contains three-tuples where the first element in the tuple is the name of the UML class. The two other elements are both sets, where *AttributeList* is the list of attributes associated with this particular class and *PointsTo* is a list of classes that this class links to. The element r is a Uniform Resource Identifier (URI) uniquely identifying the data source.

$$\begin{aligned} \textit{AttributeList} = \{\ & (\textit{att\_name}, \textit{modifier}, \textit{data\_type}) \mid \\ & \textit{modifier} \in \textit{Modifiers} \setminus \{\ *, +, \text{REQUIRED, IMPLIED}\ \} \ \wedge \\ & \textit{data\_type} \in \textit{DataTypes}\ \} \end{aligned}$$

This set describes the name, modifier and data type of each attribute in a UML class. The modifier can either be 1 or ?, describing whether or not null values are allowed for this attribute. The *data_type* element holds type information for this variable.

$$\begin{aligned} \textit{PointsTo} = \{\ & (\textit{class\_name}, \textit{link\_type}, \textit{source\_card}, \textit{target\_card}, \textit{link\_role}) \mid \\ & \textit{class\_name} \in \textit{ElementNames} \ \wedge \ \textit{link\_type} \in \textit{LinkTypes} \ \wedge \\ & \textit{source\_card}, \textit{target\_card} \in \textit{Cardinalities} \ \wedge \\ & \textit{link\_role} \in \textit{AttributeNames} \cup \{\text{NULL}\} \end{aligned}$$

This set describes a relationship from one class to another class. *class_name* is the name of the class to which a link exist, *link_type* is the type of link and *source_card* and *target_card* is the cardinality specified for the source and the target of the link, respectively. *link_role* is the role of the link. A link role is a description of a link, containing a name and a direction. In this case only the name is used since the direction is implicit (the link is always from the class holding the *PointsTo* element to the class named in *class_name*). Only association type links are given a role, since the meaning of an aggregation type link is always clear (aggregation means "contained in"), and links of this type are thus assigned NULL as *link_role*.

$$\textit{DataTypes} = \{\ \text{NUMERIC, DATE, TEXT, NULL}\ \}$$

This set describes four basic data types for attributes in the UML diagram. Data types are not applicable when generating the UML model, since the DTD contains no type information (see Section 4.7).

$$\textit{LinkTypes} = \{\ \text{AGGREGATION, ASSOCIATION}\ \}$$

The set *LinkTypes* describes the different types of relations that can exist between any pair of classes in the UML diagram.

$$\textit{Cardinalities} = \{\ 0..*, 1..*, 0..1, 1\ \}$$

Cardinalities is the set of cardinalities used to describe the quantitative relationship between elements in the UML data model. The cardinalities of a relationship are given by specifying minimum and maximum cardinalities. The cardinality "1" is used as a shorthand for "1..1".

4.3.5 Functions

This section defines functions used in the conversion from DTDs to UML diagrams. All functions are defined by specifying the domain of definition and range. A "$\rightarrow$" denotes total functions whereas a "$\hookrightarrow$" denotes a partial function.

target : *AttContentSpec* $\hookrightarrow$ *ElementNames*

This function is given an attribute specification and returns the name of an element. The function is used to pair IDREF(S) and IDs, and is only defined for the subset of *AttContentSpec* having an *att_type* of either IDREF or IDREFS.

cardinality : *Modifiers* $\cup$ *AttContentSpec* $\rightarrow$ *Cardinalities*

This function is given either an element from *Modifiers* or an element from *AttContentSpec* and returns the cardinality corresponding to the modifier or the cardinality of the attribute.

Example 4.2 Cardinalities
An attribute declared as IMPLIED has cardinality 0..1, an REQUIRED attribute has cardinality 1, an IMPLIED IDREFS attribute has cardinality 0..*, an REQUIRED IDREFS attribute has cardinality 1..*. The modifiers *, + and ? has cardinality 0..*, 1..* and 0..1, respectively.

parent : *UMLClasses* $\hookrightarrow$ *UMLClasses*

This function is given an element from *UMLClasses* and returns the parent element of this element as specified by the nesting relationship in the DTD. The function is only defined for the elements having exactly one parent.

4.4 DTD to UML Conversion Algorithm

The algorithm for generating the UML diagram has been divided into two parts. The first part, `generateUMLClass`, is a subfunction that generates one element in *UMLClasses* which corresponds to a complete class in the final UML diagram. The other part of the algorithm, `generateUML`, calls the `generateUMLClass` subfunction repeatedly until all the elements of *UMLClasses* have been generated.

The algorithms generate only valid UML diagrams. The only UML components generated by the algorithm `generateUMLClass` are classes with zero or more attributes, cardinality-specified aggregations between classes and, role- and cardinality-specified associations between classes (see Section 4.3.4 for a definition of association roles). Aggregations are always constructed in direction from parent to child and associations are always constructed in direction from the referring element to the element bearing the corresponding ID.

In general the structure of the UML diagram generated by the algorithm corresponds to the tree-structure of the DTD where a UML class is generated for each element. All elements having a parent/child relationship in the DTD are connected by an aggregation relationship in the UML diagram and ID-references between elements are modeled as associations between the corresponding UML classes.

4.4.1 GenerateUMLClass Algorithm

(1) `generateUMLClass`($x_i \in$ *DTDContent*):
(2) Generate new element $e \in$ *UMLClasses* where
(3) e[*class_name*] $= x_i$[*element_name*] $\wedge$
(4) e[*AttributeList*] $= \{(a, b, c) \mid a = x_i[\textit{AttContentSpec}_j][att\_name],$

(5) $b = \text{cardinality}(x_i[\textit{AttContentSpec}_j][\textit{modifier}]),\ c = \text{NULL} \wedge$
(6) $x_i[\textit{AttContentSpec}_j][att\_type] \notin \{$ IDREF, IDREFS $\} \} \cup$
(7) $\{ (a, b, c) \mid a =$ "value", $b = 1,\ c = \text{NULL} \wedge\ x_i$ is a leaf element$\} \quad \wedge$
(8) $e[\textit{PointsTo}] = \{(d, e, f, g, h) \mid d = x_i[\textit{ElementContentSpec}_j][\textit{element\_name}],$
(9) $e =$ AGGREGATION,
(10) $f = 1,$
(11) $g = \text{cardinality}(x_i[\textit{ElementContentSpec}_j][\textit{modifier}]),$
(12) $h = \text{NULL} \wedge$
(13) x_i is not a leaf element $\} \cup$
(14) $\{ (d, e, f, g, h) \mid d = \text{target}(x_i[\textit{AttContentSpec}_j]),$
(15) $e =$ ASSOCIATION,
(16) $f = \text{cardinality}(x_i[\textit{AttContentSpec}_j]),$
(17) $g =$ 0..*,
(18) $h = x_i[\textit{AttContentSpec}_j][att\_name] \wedge$
(19) $x_i[\textit{AttContentSpec}_j][att_type] \in \{$IDREF, IDREFS$\}\}$

The subfunction `generateUMLClass` works as follows: The function is given an element $x_i \in$ *DTDContent* and generates an element $e \in$ *UMLClasses* which is the representation of a UML class.

e[*class_name*] is set to the name of the element in the DTD to which x_i corresponds.

e[*AttributeList*] is a set of attribute names, modifiers and data types for all the non IDREF and IDREFS type attributes defined for the UML class corresponding to x_i. The data type is not applicable since it cannot be determined at this point. If the element x_i is a leaf element an attribute of name "value" is added to the class to hold the element's data. This means that all elements having their content defined as CDATA or PCDATA will have the attribute "value" to hold this character data.

e[*PointsTo*] is a set of links to each of the elements within the content specification for the element corresponding to x_i. The link type and cardinality is determined by the functions linkType and cardinality. The cardinality *source_card* for links caused by attributes of types different from IDREF and IDREFS always defined as 1 since elements in XML are always uniquely nested. The cardinality *target_card* is for attributes of type IDREF and IDREFS always set to 0..* since it in XML documents described by DTDs not can be controlled how many elements are allowed to refer to an element having an ID attribute. A link can be from one element to itself indicating a recursive definition.

The link types are either aggregations or associations. Aggregations are used whenever a link to a nested element is established (modeling the "consists of" parent-child relationship) and associations are used when modeling an ID-reference relationship between two elements. A link role is supplied for each association type link, containing the name of the IDREF or IDREFS attribute causing the link. This helps resolving the conceptual meaning of association type links in the UML diagram, provided, of course, that the IDREF and IDREFS attributes are given meaningful names in the DTD.

4.4.2 GenerateUML Algorithm

(1) `generateUML`:

(2) *UMLClasses* = $\bigcup_i$ { `generateUMLClass(`$x_i \in$ *`DTDContent`*`)` }
(3) *UMLClasses* = *UMLClasses* $\cup$ $\{r\}$

The algorithm works as follows: The set *UMLClasses* is generated by uniting the result of calling the `generateUMLClass` subfunction on every $x_i \in$ *DTDContent*. r is the URI describing the location of the XML document.

4.5 UML Post-processing

To reduce the number of classes in the UML diagram *UMLClasses* is processed by the algorithm `postProcessUML`. The number of classes is reduced by removing all leaf elements having only one parent and no association relationships. The attributes and data contained within the leaf element is moved from the leaf element to its immediate parent, thereby making it possible to remove the now empty leaf element.

4.5.1 Post-processing Algorithm

(1) `postProcessUML`:
(2) $\forall e_i \in$ *UMLClasses* where e_i is a leaf element having one parent and no references:
(3) **if** (parent(e_i)[*PointsTo*$_j$][*class_name*] == e_i[*class_name*]) $\wedge$
(4) (parent(e_i)[*PointsTo*$_j$][*target_card*] $\in$ { 1, 0..1 }) **then**
(5) $\forall a_m \in e_i$[*AttributeList*] :
(6) a_m[*att_name*] = e_i[*class_name*] $\circ$ "." $\circ$ a_m[*att_name*]
(7) **if** (parent(e_i)[*PointsTo*$_j$][*target_card*] == 0..1) **then**
(8) $\forall z_k \in e_i$[*AttributeList*] : z_k[*modifier*] = ?
(9) **endif**
(10) parent(e_i)[*AttributeList*] = parent(e_i)[*AttributeList*] $\cup$ e_i[*AttributeList*]
(11) parent(e_i)[*PointsTo*] = parent(e_i)[*PointsTo*] $\setminus$ { parent(e_i)[*PointsTo*$_j$] }
(12) *UMLClasses* = *UMLClasses* $\setminus \{e_i\}$
(13) **endif**

The algorithm works as follows: An element $e_i \in$ *UMLClasses* is a candidate for removal if e_i is a leaf element having one parent and no incoming or outgoing references. When a candidate element has been found, it is checked if the link from e_i's parent to e_i has a target cardinality of 1 or 0..1. If it has, e_i can be safely removed. The reason for only removing classes having this cardinality is that classes that are part of a "many" relationship are always modeled as separate classes in UML.

When an element e_i destined for removal has been located, the names of the attributes defined for the element are changed. The new attribute names are a concatenation of the name of the class, a dot (".") and the old attribute name (in the algorithm "$\circ$" is the concatenation operator). Finally, all the attributes from the about-to-be removed class are given a modifier corresponding to the class' cardinality and are then copied to the parent.

4.6 Example: Generating a UML Diagram from an XML DTD

This section illustrates through an example how the previous algorithms work. It is assumed that the document used in the example is located at *http://www.componentheaven.com/parts.xml*.

4.6.1 DTD

Consider the following DTD. The root element of this DTD is the `class` element.

```
<!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) >
```

4.6.2 Pre-processing the DTD

When pre-processing the DTD according to the transformations described in Section 4.2 the element content specification for element "class" is reduced to `<!ELEMENT class (ec*, device*) >`.

4.6.3 Sets Describing the DTD

The first step is to construct the sets describing the DTD (as defined in Section 4.3.2). The sets are constructed as follows:

ElementNames = { class, ec, device, unitprice, name, pincount, gatecount, textdesc, number, price, CDATA, PCDATA }

AttributeNames = { id, usedWithin, name }

DTDContent = { (class, {(ec, *), (device, *), (name, 1)}, {(name, CDATA, REQUIRED)}),
(ec, {(unitprice, 1), (pincount, 1), (gatecount, 1), (textdesc, ?)},
{(id, ID, REQUIRED), (usedWithin, IDREFS, IMPLIED),
(name, CDATA, REQUIRED)}),
(device, {(textdesc, 1), (unitprice, 1), (device, ?)},
{(id, ID, REQUIRED), (name, CDATA, REQUIRED)}),

(unitprice, {(number, 1), (price, 1)}, {}),
(pincount, {(PCDATA, 1)}, {}),
(gatecount, {(PCDATA, 1)}, {}),
(textdesc, {(PCDATA, 1)}, {}),
(number, {(PCDATA, 1)}, {}),
(price, {(PCDATA, 1)}, {}) }

The set *DTDContent* is a three-tuple of the element name, the element's content specification and the element's attribute specification. The second and third element in the tuple, which are the sets *ElementContentSpec* and *AttributeContentSpec* are defined in Section 4.3.2.

4.6.4 Sets Describing the UML Diagram

When the algorithm `generateUML` is applied to the sets describing the DTD, it calls the subfunction `generateUMLClass` on every element $x_i \in$ *DTDContent*. For each x_i an element $e_i \in$ *UMLClasses* is generated. Thus when called on the "device" element from *DTDContent* (hereafter denoted x_{device}) the subfunction will generate an element which is hereafter denoted e_{device}.

e_{device} is constructed as follows:

e_{device}[*class_name*] = x_{device}[*element_name*] = "device"
e_{device}[*AttributeList*] = { (id, 1, NULL), (name, 1, NULL) }
e_{device}[*PointsTo*] = {("textdesc", AGGREGATION, 1, 1, NULL),
("unitprice", AGGREGATION, 1, 1, NULL)}
("device", AGGREGATION, 1, 0..1, NULL)}

The e_{device} class has two attributes defined, corresponding to the attribute "id" of type ID and the attribute "name" of type CDATA defined for this element in the DTD. The element has three links to the three classes "textdesc", "unitprice", and "device". The links are all aggregation type links since they are all children of the "device" element in the DTD. The link from e_{device} to "device" is a link from e_{device} to itself, indicating that "device" is recursively defined. All aggregation type links are given a *link_role* of NULL since there is no confusion about the meaning of aggregation type links.

As another example, the *UMLClasses*-element e_{ec} corresponding to the more complex XML element "ec" is specified below.

e_{ec} = ("ec", {(id, 1, NULL), (name, 1, NULL)},
{("unitprice", AGGREGATION, 1, 1, NULL),
("pincount", AGGREGATION, 1, 1, NULL),
("gatecount", AGGREGATION, 1, 1, NULL),
("textdesc", AGGREGATION, 1, 0..1, NULL),
("device", ASSOCIATION, 0..*, 0..*, "usedWithin")})

As in the previous example an aggregation type link exist for all children of this element. In addition, an association type link exist to the "device" class because of the IDREFS attribute

"usedWithin" declared for element "ec". This link is given the *link_role* "usedWithin" which is the name of the IDREFS attribute causing the link.

As an example of the specification of a leaf element, the element $e_{gatecount} \in UMLCLasses$ is shown below.

$e_{gatecount}$ = ("gatecount", {("value", 1, NULL}, {})

The attribute "value" defined for the leaf element $e_{gatecount}$ is used to hold the character type data contained within the "gatecount" element. Since a leaf element is defined as being any element having a content specification of some character data type (e.g. CDATA or PCDATA) all leaf elements will carry the "value" attribute to hold this data.

As defined in Section 4.3.4, *UMLCLasses* also contains the URI of the data source, which in this case is the URL *http://www.componentheaven.com/parts.xml*.

4.6.5 UML Diagram

A UML diagram describing the DTD can be built directly from the sets describing the UML diagram. The UML diagram for the DTD in this example is shown in figure 2. Notice the labeled arrow on the association type link from "ec" to "device", which is a representation of the *link_role* element associated with this link. The labeled arrow indicates the direction and name of this link, making it easier to understand the relationship between the two classes.

4.6.6 UML Post-processing

The UML diagram in figure 2 has one class for each element specified in the DTD. To reduce the total number of classes the `postProcessUML` algorithm is applied to the *UMLClasses* set. The UML diagram resulting from the application of `postProcessUML` is shown in figure 3. As can be seen from the figure, four classes have been removed by the post processing algorithm.

4.7 Determining Attribute Data Types

In the generation of the UML diagram a set of attributes is associated with each of the classes as described in Section 4.3.4. Since the UML diagram is generated directly from the XML DTD it is not possible to determine data types for the attributes at that point, and they are therefore typed with NULL. However, before generating an OLAP DB, data types has to be determined for the attributes related to the post-processed UML diagram.

It is not the intention of this section to provide a final solution of how type information is assigned to attributes. Instead, some of the problems associated with type determination is presented along with brief suggestions on how to resolve these difficulties.

4.7.1 XML Type Information

One of the problems in determining type information for the attributes, is that XML 1.0 is a typeless language, so no data type declaration exists in the DTD or in the XML document. [28]

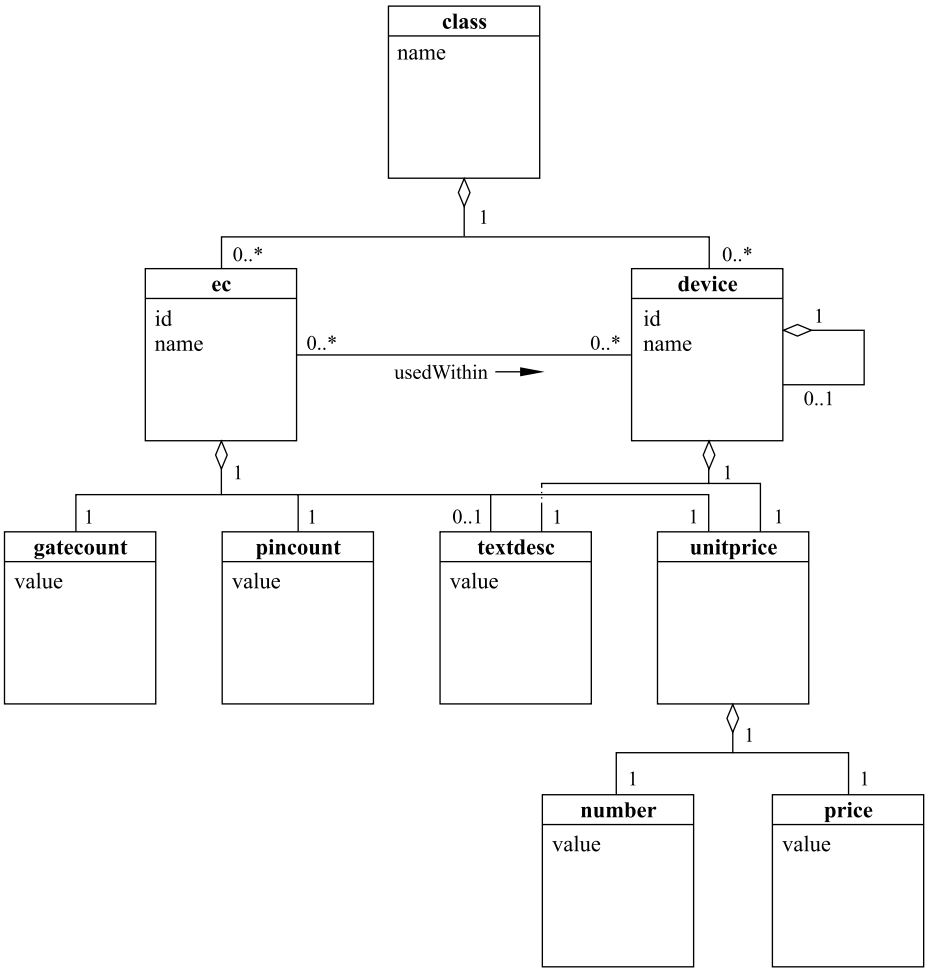

Figure 2: UML diagram as produced by the `generateUML` algorithm

The type space for XML is simply string values [28], but this information is to vague to be of any practical use in an OLAP system (e.g. when wanting to use aggregation functions other than count). Therefore the type space is extended as described in Section 4.3.4 by the set *DataTypes* (having members NUMERIC, DATE, etc.).

4.7.2 Type Determination Methods

One way to assign type information to the attributes is to sample from the XML document. For each of the attributes in the UML diagram a number of samples can be done from the data contained within the corresponding XML tags, thereby making it possible with a certain probability to make a reasonable type determination.

Example 4.3 Sampling
As an example of this approach consider the following fragment of an XML document: `<price>123</price>`. By using sampling, the attribute price could be assigned the type NUMERIC.

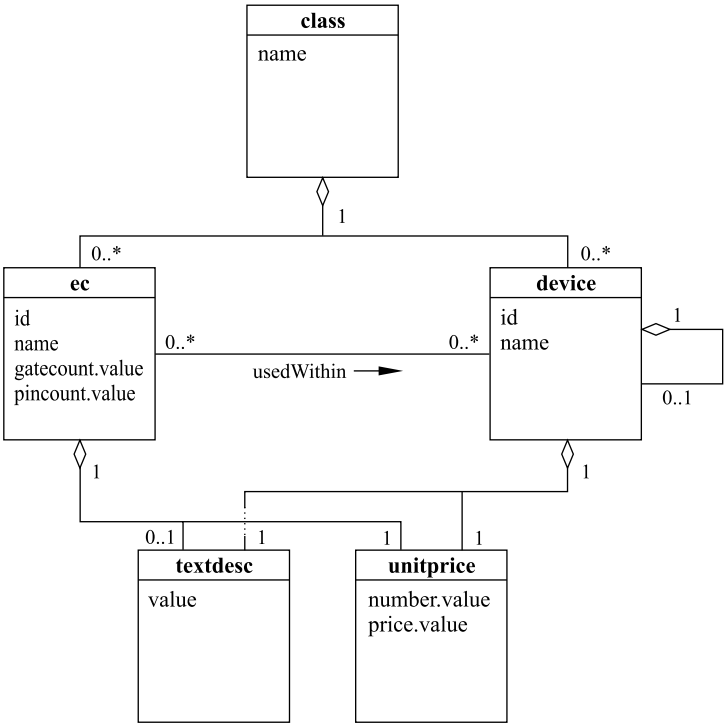

Figure 3: UML diagram after post-processing

The designer can also assist in determining types. It is fair to assume that the designer has some a priori knowledge about the data used for the analysis. By presenting the attributes of concern, the designer can make type assignments to these attributes based on a priori knowledge.

Another way to resolve types is to simply provide a lexical analysis of the element declarations in the DTD, in order to make qualified guesses of types based on certain keywords.

Example 4.4 Lexical Analysis
By using the approach based on lexical analysis an element named “amount” defined as PCDATA could be assumed to contain data of type NUMERIC.

Of course, each of these methods can be combined in order to get the best possible assignment of types. As an example of a combined approach consider the following DTD fragment along with a fragment of an XML document conforming to the DTD:

DTD:	**XML document:**
`<!ELEMENT date (#PCDATA) >`	`<date>01/01/01</date>`

By applying the sampling method and lexical analysis of element definitions the element `date` could be assigned the type DATE.

However, even though these methods can be used to provide type bindings for the attributes, complications exists. As described in Section 3.1, the only data initially fetched from the data source is the DTD. Not knowing the actual structure or size of the data makes it hard to make a sound statement about the statistical value of the sampling. Furthermore, having only a query

interface to the data makes it impossible to sample at random, since determining a random position in the XML document is not possible. A way of determining types unambiguously is to read the entire XML document. This is not considered an option due to possibly very large documents.

It is beyond the scope of this paper to go further into the problems and solutions associated with type determination. Instead a function determineType is assumed, that assigns types from the set *DataTypes* to each of the elements in the set *AttributeList* associated with each of the UML classes.

5 Construction of Multidimensional Model

This section describes the construction of a multidimensional model from 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 multidimensional model 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 illustrated graphically, 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. When based on XML data though, some structural properties inherent to XML aids the design process. This section formally defines 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.

The constructed UML snowflake diagram is not a foundation for a physical integration of the data elements into a traditional data warehouse [10, 11], rather a structural description of a multidimensional understanding of the data elements involved. Instead of creating a physical (materialized) data warehouse, data are fetched on-demand from the respective sources and fed to the OLAP system when requested.

The section is concluded with an example, showing the construction of a UML snowflake diagram based on data from various XML documents and relational data sources.

5.1 Query Language for UML diagrams

In order to precisely describe properties of the UML snowflake diagram, a language for querying UML diagrams is needed. 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 [4]. 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.

The UML diagram in Figure 4 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 4 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.

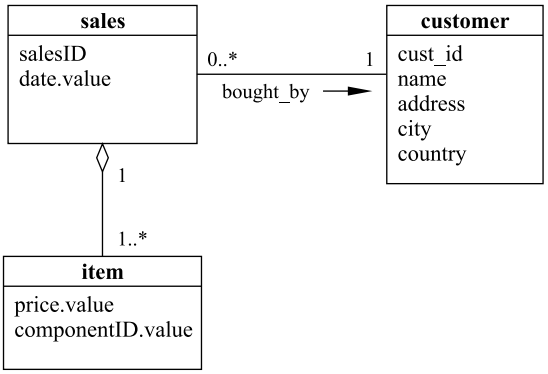

Figure 4: UML diagram

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 4. 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. [2]

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.

5.2 UML Snowflake Diagram

This section defines the UML snowflake diagram which is a multidimensional model 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 5.1 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 $m > n$ 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 $n \geq 0$ from $class_i$ is termed a *subgraph*. No edges exists between 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 Section 4.4, 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.

Example 5.1 UML Snowflake Diagram
As an example of a UML snowflake diagram consider Figure 5.

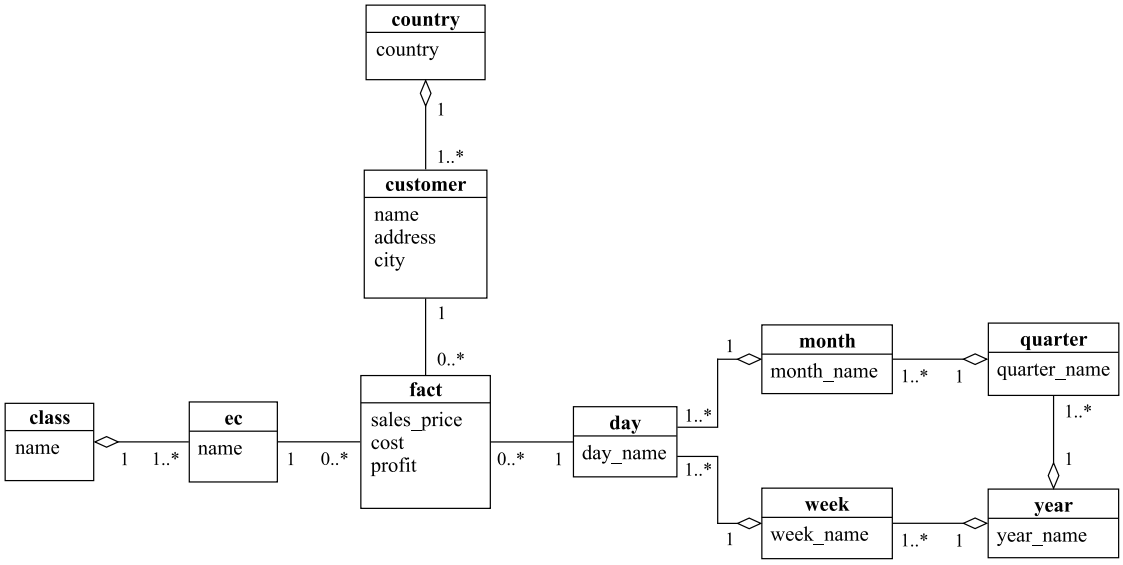

Figure 5: UML snowflake diagram

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.

5.2.1 Sets Describing the UML Snowflake Diagram

This section defines 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*) |
att_type ∈ *DataTypes* ∧ *modifier* ∈ { ?, 1 } }

This set describes the name, data type, modifier, Attribute Identifier (AID), formula, and link path for each attribute associated with a class. *att_name* is a unique name identifying the attribute.

att_type is the data type of the attribute, as determined by the function determineType described in Section 4.7. *modifier* indicates whether or not null values are allowed for the attribute.

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* (as defined in Section 4.3.4), 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.

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 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 [14]).

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

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 in Section 5.2.3 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. *link_role* is used for naming an association type link as described in Section 4.3.4.

PointsTo = { (*class_name, link_type, source_card, target_card*) |
source_card ∈ { 0..*, 1..* } ∧ *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*. *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 5.1. *target_card* is always set to 1.

Example 5.2 UML Snowflake Diagram Sets
Consider the UML snowflake diagram from Example 5.1 depicted in Figure 5.

The attribute list for dimension class "ec" is defined as:

$$\begin{aligned} \mathit{AttributeList}_{ec} = \{ &(\text{id, NUMERIC, 1, } \mathit{URL}_{component}\text{:class/ec.id, NULL, NULL}), \\ &(\text{name, TEXT, 1, } \mathit{URL}_{component}\text{:class/ec.name, NULL, NULL}) \} \end{aligned}$$

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 $\mathit{URL}_{component}$: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:

$$\begin{aligned} \mathit{PointsTo}_{fact} = \{ &(\text{ec, ASSOCIATION, 0..*, 1}), (\text{customer, ASSOCIATION, 0..*, 1}), \\ &(\text{day, ASSOCIATION, 0..*, 1}) \} \end{aligned}$$

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

5.2.2 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.

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 5.3 Creating a Fact Class
Consider the UML diagram in Figure 6 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

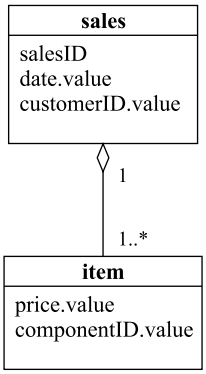

Figure 6: UML diagram describing sales

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.

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 “sales” is used as the foundation for the fact class. Attributes from class “item” is added to the “sales” class, yielding the following attribute list for the fact class (some of the attributes have been renamed):

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

The constructed fact class is illustrated in Figure 7 containing the measure attribute “sales_price”.

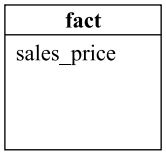

Figure 7: UML diagram describing the fact class

5.2.3 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 only contain attributes describing who bought the product, the type of product sold, and the amount of money the product was sold for, the attribute set for the class has to be extended with an attribute stating the amount paid by the retailer for the product in order to calculate the profit for the sale.

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.

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

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. So at query time a way to

explicit relate the data used in establishing a connection between class T and S_n is needed.

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. That a many-to-one relationship really exist between the classes has to be checked at query time when the actual matching of attributes is performed. How to ensure the required many-to-one relationship is described in Section 6.2.

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 5.4 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 8 (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 8, 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 8 (b) has been constructed by the retailer.

As it can be seen from Figure 8, 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 8 as solid lines from diagram (a) through diagrams (b) and (c). The graphical illustration of path establishment and attribute retrieval (dotted line in Figure 8) 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 in Section 5.2.1, 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 8 are located at URL_{fact}, $URL_{supplier}$ and $URL_{component}$, respectively.

(cost, NUMERIC, 1, $URL_{component}$:class/ec/unitprice.price.value, `null`,

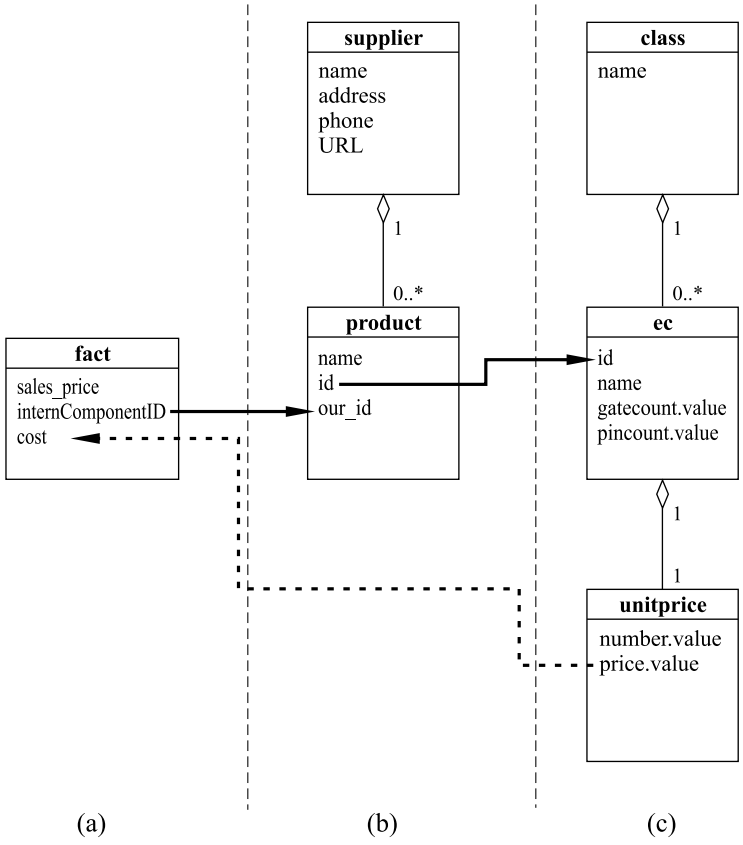

Figure 8: UML diagrams (a), (b) and (c)

(internComponentID, $URL_{supplier}$:supplier/product.our_id,
$URL_{supplier}$:supplier/product.id, $URL_{component}$:class/ec.id))

5.3 Constructing Multidimensional Models from XML Documents

When constructing multidimensional models like the UML snowflake diagram, some properties inherently connected to XML data can be exploited. The generel structure of XML data is a hierachical 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 af the leaf level of the XML tree. [1, 20] 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` as described in Section 4.4.2 is designed in a way 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 multidi-

mensional understanding of a hierarchy where the ALL-element (corresponding to the root in the XML document) defines the most general case [7].

However, a few special cases occurring in some XML designs needs special attention when constructing an OLAP DB from 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 two ways. One way is to "split up" the overlying hierarchy in as many parts as the number of parents to the element, and then choose what part of the hierarchy to use. As an example of this, consider Figure 3 in Section 4.6.6. 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".

If the purpose of the OLAP DB is to analyse sales of different products, both components and devices, the fact table and the product dimension has to be split by product type. The splitting of heterogeneous fact tables and dimensions is a common modeling issue and will not be explained in detail here. Instead, see [11] for a detailed explanation.

The other way of resolving the problem of multiple parents is to 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 decedent 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 in the UML diagram represented 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. Again, 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 3, Section 4.6.6) indicating that the component is used to build that particular device, is of no relevance to the analysis and therefore ignore the reference.

The other 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 case of an optional IDREF attribute or an IDREFS attribute this constraint is clearly not satisfied. How to ensure the required many-to-one relationship at query time is described in Section 6.2.

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. A hierarchy where the same level of generalization appears at different levels in the hierarchy does not make sense.

Again, as with the problem of ID-references between elements, the designer could choose to ignore the recursive definition by setting the maximum recursive depth to zero.

Whether or not the solution of ignoring the recursive definition is applicable depends on the domain of the source data and the purpose of the OLAP DB. Based on knowledge of the domain, the designer must choose how to handle recursively defined elements, either by ignoring the definition or by using parts of the recursively defined hierarchy for constructing new hierarchies or dimensions ad hoc.

As claimed in [20], 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.

5.4 Example: Construction of a UML Snowflake Diagram

The following example illustrates the construction of a UML snowflake diagram according to Definition 5.1. The UML snowflake diagram constructed is the diagram in Example 5.1 in Section 5.2, 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. Each of the data sources are described in the following sections.

5.4.1 Sales Document

The UML diagram derived from the sales document DTD, which describes sales of components is depicted in Figure 9. The diagram is the same as described in Example 5.3. It is assumed that the XML document is located at URL_{sales}.

5.4.2 Component Document

The component document describing the supplier's products used in this example is the same as the one described in Section 4.6. The UML diagram is depicted in Figure 10. It is assumed that

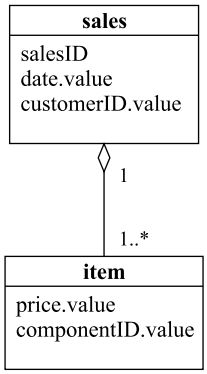

Figure 9: UML diagram derived from the sales document DTD

the XML document is located at $URL_{component}$.

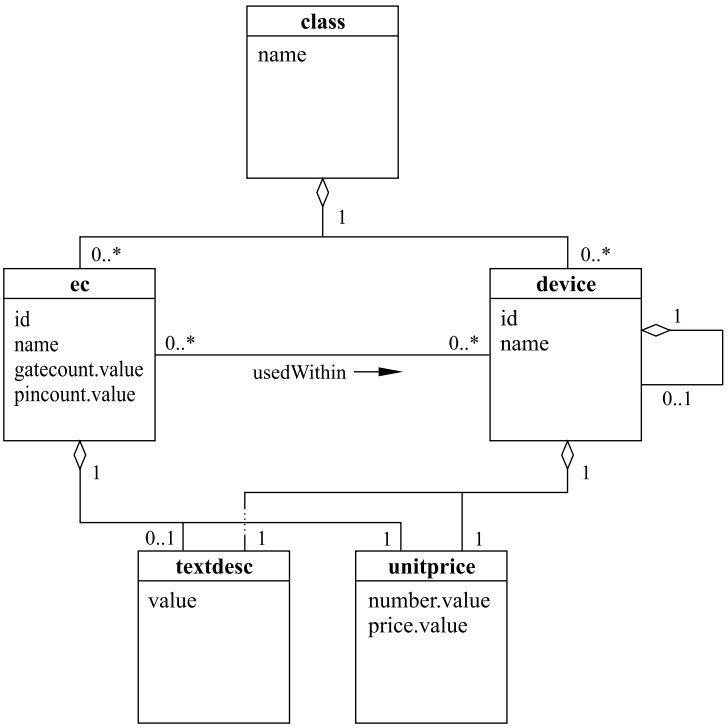

Figure 10: UML diagram derived from the component document DTD

5.4.3 Customer Data

Data describing customers is stored in a relational database. As assumed in Section 3.2 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,

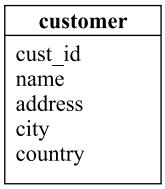

Figure 11: UML diagram describing customer data

the address (street and number), the city, and the country in which the customer lives.

5.4.4 Construction of UML Snowflake Diagram

This section describes 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 four sections describe the construction of the fact class and the three dimensions.

Fact Class

The content of the fact class is primarily composed of data from the sales document (as described in Example 5.3). 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 5.4 in Section 5.2.3. The attribute "profit" is calculated by subtracting "cost" from "sales_price".

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

$$
\begin{aligned}
AttributeList_{fact} = \{\ & (\ \text{customerID, NUMERIC, 1, } URL_{sales}\text{:sales.customerID.value,} \\
& \quad \text{NULL, NULL}\ ), \\
& (\ \text{date, DATE, 1, } URL_{sales}\text{:sales.date.value, NULL, NULL}\ ), \\
& (\ \text{internComponentID, NUMERIC, 1,} \\
& \quad URL_{sales}\text{:sales/item.componentID.value, NULL,} \\
& \quad (\ (\text{item, AGGREGATION, NULL})\ )\ ), \\
& (\ \text{sales\_price, NUMERIC, 1, } URL_{sales}\text{:sales/item.price.value, NULL,} \\
& \quad (\ (\text{item, AGGREGATION, NULL})\ )\ ), \\
& (\ \text{componentID, NUMERIC, 1, } URL_{supplier}\text{:supplier/product.id, NULL,} \\
& \quad (\text{internComponentID, } URL_{supplier}\text{:supplier/product.our\_id})\ ) \\
& (\ \text{cost, NUMERIC, 1, } URL_{component}\text{:class/ec/unitprice.price.value, NULL,} \\
& \quad (\text{componentID, } URL_{component}\text{:class/ec.id})\ ), \\
& (\ \text{profit, NUMERIC, 1, NULL, sales\_price - cost, NULL}\ )\ \}
\end{aligned}
$$

$PointsTo_{fact}$ is the set of classes that "fact" is connected to through associations with cardinalities as defined in Definition 5.1. 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.

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

Component Dimension

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 hierarchical organised dimension, thereby exploiting the hierarchically structure of the XML document.

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

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

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.

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

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

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

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 data described by the UML diagram in Figure 11 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 generel than a specific address.

The "customer" level contains the attributes "cust_id", "name", "address", and "city". The "country" level contains the attribute "country".

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.

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

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.

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

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

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

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 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 5.1, the time dimension contains two hierarchies. The dimension has two hierarchies since weeks do not roll up to neither months or quarters.

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

$$\begin{aligned} AttributeList_{day} = \{ &(\text{day\_name, TEXT, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \\ &(\text{date, DATE, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \\ &(\text{week\_id, NUMERIC, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \\ &(\text{month\_id, NUMERIC, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \} \\ PointsTo_{day} = \{ &(\text{week, AGGREGATION, 1..*, 1}), (\text{month, AGGREGATION, 1..*, 1}) \} \end{aligned}$$

$$\begin{aligned} AttributeList_{week} = \{ &(\text{week\_name, TEXT, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \\ &(\text{week\_id, NUMERIC, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \\ &(\text{year\_id, NUMERIC, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \} \\ PointsTo_{week} = \{ &(\text{year, AGGREGATION, 1..*, 1}) \} \end{aligned}$$

$$\begin{aligned} AttributeList_{month} = \{ &(\text{month\_name, TEXT, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \\ &(\text{month\_id, NUMERIC, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \\ &(\text{quarter\_id, NUMERIC, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \} \\ PointsTo_{month} = \{ &(\text{quarter, AGGREGATION, 1..*, 1}) \} \end{aligned}$$

$$\begin{aligned} AttributeList_{quarter} = \{ &(\text{quarter\_name, TEXT, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \\ &(\text{quarter\_id, NUMERIC, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \\ &(\text{year\_id, NUMERIC, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \} \\ PointsTo_{quarter} = \{ &(\text{year, AGGREGATION, 1..*, 1}) \} \end{aligned}$$

$$\begin{aligned} AttributeList_{year} = \{ &(\text{year\_name, TEXT, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \\ &(\text{year\_id, TEXT, 1, } URL_{sales}\text{:sales.date, NULL, NULL}) \} \end{aligned}$$

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 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.

$$\begin{aligned} SnowflakeDiagram = \{ & (\text{fact}, AttributeList_{fact}, PointsTo_{fact}), \\ & (\text{ec}, AttributeList_{ec}, PointsTo_{ec}), \\ & (\text{class}, AttributeList_{class}, PointsTo_{class}), \\ & (\text{customer}, AttributeList_{customer}, PointsTo_{customer}), \\ & (\text{country}, AttributeList_{country}, PointsTo_{country}), \\ & (\text{day}, AttributeList_{day}, PointsTo_{day}), \\ & (\text{week}, AttributeList_{week}, PointsTo_{week}), \\ & (\text{month}, AttributeList_{month}, PointsTo_{month}), \\ & (\text{quarter}, AttributeList_{quarter}, PointsTo_{quarter}), \\ & (\text{year}, AttributeList_{year}, PointsTo_{year}) \} \end{aligned}$$

The snowflake diagram is depicted in Figure 12.

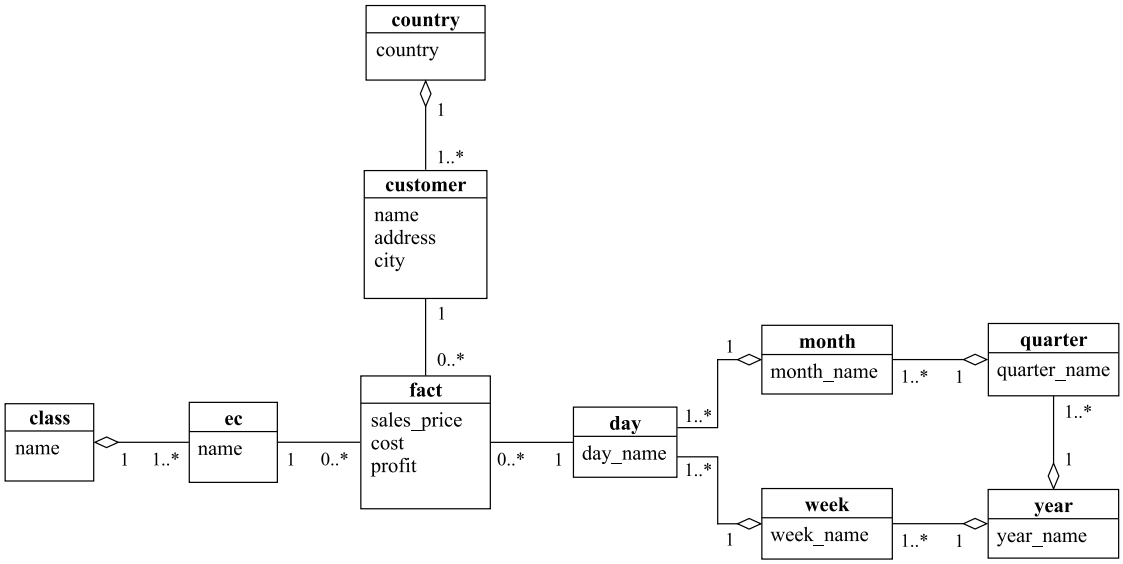

Figure 12: UML Snowflake Diagram

This concludes the example, illustrating how an OLAP DB can be constructed from different XML documents and relational data sources.

6 Retrieving Source Data

This section discusses issues related to the retrieval of the data described by the UML snowflake diagram.

In order to retrieve the data contained within the different data sources used in the creation of the UML snowflake diagram, a mapping between the queries issued by the OLAP system and the language native to the sources must exist. Two different XML query languages, a well-supported language of low expressive power and a poorly-supported language of high expressive power are compared.

6.1 Querying OLAP Data

The UML snowflake diagram constructed in Section 5 is presented as a set of relational tables (a snowflake schema) to the OLAP system. The conversion from UML to relational tables is trivial due to the structure of the UML diagram. Each UML class corresponds to one relational table, where primary/foreign keys relate the tables. Notice that due to the cardinalities specified for the relationships between the UML classes, referential integrity is required for all foreign key/primary key relationships.

6.1.1 Types of Queries Issued by the OLAP System

The types of queries issued by the OLAP system to the relational source can be categorized into two different categories called "Standard Template Queries" [11]. These categories include single table browsing and multi-table joins that joins dimensional tables and the fact table while performing grouping and aggregation. This means that the SQL queries issued by the OLAP system are always on the form

SELECT $a_1, a_2, ..., a_n$
FROM A
WHERE *predicate*

for single table browse queries, while multi-table join queries are on the form

SELECT $f_1, f_2, ..., f_m$
$d_{1_1}, d_{1_2}, ..., d_{1_n}$,
$d_{2_1}, d_{2_2}, ..., d_{2_k}$,
...,
$d_{p_1}, d_{p_2}, ..., d_{p_q}$,
aggregate(*attribute*$_i$)
FROM $F, D_1, D_2, ..., D_p$
WHERE *predicate*
GROUP BY(*attribute*$_j$)

These are the types of queries the OLAP system uses to retrieve data. These queries are to be transformed into a series of queries issued to the different data sources (either XML or relational) in order to retrieve the necessary data.

6.2 Querying Multiple Data Sources

Every table in the snowflake schema potentially contains data from a number of different relational data sources and a number of different XML documents. This means that several queries must be constructed in order to fetch the data. When executing the queries, two important issues arise: In which order to execute the queries and what to do if the requested data is either missing or differ from what was expected. These issues are discussed in the following two sections.

6.2.1 Query Order

The attributes in a UML class are of three different types, hereafter referred to as type one, two, and three attributes.

- Type one attibutes are attributes having no *link_path* and no formula associated, i.e. attributes native within the class not dependent on any calculations (see Section 5.2.1 for a description of *link_path* and calculated attributes)
- Type two attributs are attributes having a *link_path* but no formula
- Type three attributes are attributes defined by a formula

These three different kind of attributes must be fetched in the correct order. As described in Section 5, the set of type one attributes is a non-empty set of attributes originating from one class in one data source. This set of attributes is used as a basis for adding other attributes, as described in Section 5.2.3. Therefore, since the type two and possibly the type three attributes depends on the type one attributes, type one attributes must be fetched first.

The type two attributes are added by the establishment of a link path as described in Section 5.2.3. Therefore, they are to be fetched after the type one attributes.

Finally, when both the type one and type two attributes are fetched, the type three attributes, the calculated attributes, can be created by applying their formulas to the already fetched type one or two attributes.

6.2.2 Data Integrity

When data is fetched, issues as missing or "wrong" data, absence of referential integrity, and non-responding servers may arise. Compared to fetching data from an in-house database, these problems are more likely to appear due to the involvement of several autonomous data sources accessed via the Web. Below is a brief discussion of how these issues can be resolved at query time.

Missing or Duplicate Data

When retrieving data, data elements may be missing or duplicate. As an example, in an XML document describing components, even though a "name" attribute is required for each component element, it is possible to define the attribute as `name=""` (the empty string). As another example, if one "pincount" element is mandatory, the document is still valid if the element is defined as either `<pincount></pincount>` or `<pincount> 23 432 23 1 432 </pincount>`, stating either zero or several pincounts, respectively.

The problem of missing data can be solved by extending the UML snowflake diagram with a number of default values for each attribute. The default values are set by the designer, and are used whenever the corresponding attribute is missing or unusable. As an example, if doing analysis of component sales, a default value for the attribute stating the name of a component could be set to “unnamed component”. By allowing several default values for each attribute, an appropriate default value can be chosen depending on what caused the missing attribute, e.g. lack of referential integrity, wrong data type, multiple values etc.

If referential integrity between two relations is violated, two solutions are possible for solving the problems at runtime. One approach is to use a specified default values instead of the missing or duplicate data, as described above. Another approach is to discard the data values causing the violation of referential integrity, or, in the case of duplicate data (a many-to-one relationship actually being a many-to-many relationship), simply to choose at random between the set of referenced data elements.

As an example, if a component id in the fact table is not found in the product dimension, referential integrity is clearly violated. Instead of the missing dimensional data, the default values specified for each attribute in the dimension are used. The second approach is to simply ignore the tuple in the fact table, excluding that tuple from the analysis. Which solution to choose depends on the nature of the analysis and is therefore to be chosen by the designer.

When retrieving data, the system should count the number of times default values are used instead of data values and the number of times data is discarded, alerting the user and possibly terminating the process if the number exceeds some predefined threshold. If the value remains below the limit, a report should be generated stating this count.

Data of Wrong Type

As described in Section 4.7, type determination of elements in XML 1.0 documents is inherently error prone. The problem of wrong data types is similar to the problem of missing data, since a data value having a type different from the expected can be considered a missing value. The solution is either to use default values, ignore the data element, or to terminate the process.

Servers Being Down

When accessing a number of different servers via the Web there is a possibility of one or more servers being down. If a server is down, it is most likely that the absence of an entire data source will render the entire analysis useless, if default values are used instead of the missing values. The only way of “solving” this problem is to raise a master-exception terminating the process and alerting the user.

6.3 Choice of XML Query Language

The choice of language for querying XML data has an impact on both efficiency and simplicity. Two different query languages, XPath and Quilt are compared.

6.3.1 XPath

XPath is a language for addressing parts of an XML document and is a W3C Recommendation [30]. The input to XPath is an XML document, and the output is a fragment of that XML

document, where order and structure is preserved. XPath exploits the hierarchical structure of XML by providing a path notation for navigating through the XML document. The path expressions used by XPath is similar to the path expressions used when addressing a file in a conventional filesystem. As an example, to select all children to the `item` elements in the sales document described in in Section 2.1, the XPath expression is

```
/salesDB/sales/item/*
```

As when specifying paths in a filesystem, a leading slash ('/') means an absolute path, whereas the absence of a leading slash means a path relative to some position, which in XPath is called a "context-node".

In addition to path expressions, XPath also define predicates for matching an element's type, value, and position. As an example, to select all sales made to the customer having id "2347", the XPath expression is

```
/salesDB/sales[customerID='2347']
```

stating that all `sales` elements, children of `salesDB`, having a child element called `customerID` with a value of "2347" are to be selected.

As can be seen from the above examples, XPath uses a very compact syntax. A short syntax has been chosen to enable the use of XPath as a part of an URL for addressing a specific part of an XML document. Since the main purpose of XPath is to address parts of XML documents, many constructs normally found in query languages are absent. As an example, XPath has no support for joins, group-bys, or order-bys, making several types of queries tedious and ineffective. Furthermore, XPath lacks the ability for restructuring XML, making selection of multiple elements difficult. In the case where two elements, a and b where b is a descendant to a are to be selected, this can only be done by either selecting all elements in the path from a to b or by issuing several queries to retrieve the elements one by one.

6.3.2 Quilt

Quilt is a full-blown XML query language for querying multiple XML data sources [5]. Quilt is a full-blown query language in the sense that it supports all constructs found in languages such as SQL or OQL, including joins, group-bys, order-bys, and aggregate functions. The input and output to Quilt are XML documents, fragments of XML documents or collections of several XML documents.

The syntax of Quilt is based on the syntax of XPath, XML-QL [6], and SQL [14]. From XPath, the notation of a path is used, and from XML-QL the use of variables for binding values and the use of a RETURN clause (corresponding to the XML-QL CONSTRUCT clause) is adopted. The structure of a Quilt query is based on the "FLWR expression" as opposed to the "SELECT-FROM-WHERE" statement used in SQL. A FLWR expression is constructed from FOR, LET, WHERE and RETURN clauses which must appear in a specific order.

As an example, to illustrate a natural join between three XML documents, the following query lists the supplier's name of every distinct component sold by the retailer. The query joins the retailer's and the suppliers component id through the mapping document. The documents used are

the documents sales.xml, mapping.xml, and products.xml, as described in Section 2.1 and Section 2.2.

```
FOR $intern_id IN distinct(document("sales.xml")
                           /salesDB/sales/item/componentID),
    $extern_id IN document("mapping.xml")
                  /supplierDB/supplier/product/
                       @id[@our_id = $intern_id],
    $name IN document("products.xml")
             /products/class/ec/@name[@id = $extern_id]
RETURN
    <name>
      $name
    </name> SORTBY($name)
```

The query binds the variable "$intern_id" to a component id in the sales document, uses the value of this variable to bind the variable "$extern_id" to the corresponding id in the mapping document, and finally uses the value of "$extern_id" to find the name of the corresponding component in the product document. In the RETURN clause, a new document is built, containing the component names in alphabetical order. For more examples of Quilt queries, refer to [5].

Opposed to XPath, Quilt directly supports all the constructs used in the SQL queries described in Section 6.1.1. Consider the case where n XML documents are used to create a fact table and one dimension. A multi table join, joining the fact table and a dimension would, due to the lack of inter-document joins, require at least n different XPath queries which has to be executed in correct order. Furthermore, if grouping and aggregation is used, all the data contained within the n different data sources has to be fetched and grouped afterwards. If Quilt is used, only one query is necessary, and a less amount of data has to be transfered since grouping and aggregation is done locally on the servers containing the data.

Quilt enables an easier and more efficient way of querying XML data compared to XPath. Unfortunately, while XPath is a standard widely supported, Quilt is not supported by any commercial product, making it a poor choice of query language. At the time of writing, no standard high-level XML query language exist.

The World Wide Web Consortium (W3C) is currently working on defining a standard XML query algebra [29], having the same expressive power and features as Quilt. When an XML query algebra becomes a standard, a formal mapping between the SQL queries issued by the OLAP system and the XML algebra can be used as a basis for transforming SQL queries into the syntax of some XML query language.

7 Conclusions 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 algorithms and techniques that enable existing OLAP tools to exploit XML and relational data, without requiring physical integration of data.

This paper propose a multidimensional model, the UML snowflake diagram, constructed from multiple XML and/or relational data sources, thereby making OLAP analysis possible by logically integrating data from semistructured and/or structured sources. The logical integration of XML documents is accomplished by "disguising" XML data as relational data, where low time complexity algorithms transforms the logical structure of an XML document (the DTD) into the widely used high-level semantic data model UML. The UML diagraming method is used for describing and visualizing the logical structure of XML documents, easing the design of the OLAP DB.

The relational structures generated from the UML snowflake diagram is populated by transforming queries from the OLAP tool into an appropriate semistructured query language used to query the XML data sources. This paper does not present algorithms for transforming OLAP queries into semistructured queries, since no suitable standardized query language exist.

The implementation of a prototype utilizing the aspects described in this paper is the next immediate step to be taken. Furthermore, it will be investigated how query handling techniques can be efficiently implemented. The greatest disadvantage of logical integration as opposed to physical integration is query performance. Therefore it is of out-most importance that the translation from OLAP queries into semistructured queries is done efficiently with effective query optimization. Another research direction is to consider the issue of caching data. Storing higher-level summaries of the data can speed up query processing considerably [19]. If caching of aggregates is employed, the *data explosion problem* [23, 25] needs to be taken into account, e.g. by employing *practical* pre-aggregation [8]. Furthermore, if XML Schema advances to a W3C Recommendation it would be interesting to consider using this 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 could also be investigated.

References

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

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

[3] Bonifati, Angela et. al. Comparative Analyses of Five XML Query Languages, *Special Interest Group on Management of Data Record 29(1): 68-79 (2000)*

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

[5] Chamberlin, Don et. al. Quilt: An XML Query Language for Heterogeneous Data Sources *WebDB (Informal Proceedings) 53-62 (2000)*

[6] Deutsch, A. et. al. A Query Language for XML, *Proceedings of the International World Wide Web Conference, WWW8*, Toronto, Canada, May 1999

[7] Gray, Jim 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)*

[8] Harinarayan, V. et. al. Implementing Data Cubes Efficiently, *Proceedings of Special Interest Group on Management of Data*, pp. 205-216, 1996

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

[10] Kimball, Ralph et. al. *The Data Warehouse Lifecycle Toolkit - Expert Methods for Designing, Developing, and Deploying Data Warehouses*, John Wiley & Sons, Inc., 1998

[11] Kimball, Ralph *The Data Warehouse Toolkit - Practical Techniques for Building Dimensional Data Warehouses*, John Wiley & Sons, Inc., 1996

[12] Lahiri, T et. al. Ozone: Integrating Structured and Semistructured Data, *Proceedings of the Seventh International Conference on Database Programming Languages*, Kinloch Rannoch, Scotland, September 1999.

[13] 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

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

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

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

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

[18] Pedersen, Torben Bach et. al. Extending OLAP Querying To External Object Databases, *Proceedings of the Ninth International Conference on Information and Knowledge Management*, pp. 405-413, 2000

[19] 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

[20] Pinnock, Jon et. al. *Professional XML*, Wrox Press, 2000

[21] Rafanelli, M. et. al. STORM: A Statistical Object Representation Model, *Proceedings of the Fifth Conference on Statistical and Scientific Database Management*, pp. 14-29, 1990

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

[23] Shukla, A et. al. Storage Estimation for Multidimensional Aggregates in the Presence og Hierarchies, *Proceedings of Very Large Databases*, pp. 522-531, 1996

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

[25] The OLAP Report *Database Explosion*, http://www.olapreport.com/DatabaseExplosion.htm. Current as of January 2nd, 2001

[26] Thomsen, E. et. al. *Microsoft OLAP Solutions*, John Wiley & Sons, Inc., 1999

[27] Thomsen, E. *OLAP Solutions: Building Multidimensional Information Systems*, John Wiley & Sons, Inc., 1997

[28] World Wide Web Consortium *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation, http://www.w3.org/TR/2000/REC-xml-20001006, Oct. 6 2000. Current as of December 20th, 2000

[29] World Wide Web Consortium *The XML Query Algebra*, W3C Working Draft, http://www.w3.org/TR/query-algebra, Dec. 4 2000. Current as of January 6th, 2001

[30] World Wide Web Consortium *XML Path Language (XPath) 1.0*, W3C Recommendation, http://www.w3.org/TR/1999/REC-xpath-19991116, Nov. 16 1999. Current as of December 18th, 2000

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