AALBORG UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Title:

Using XML Data in OLAP Queries

Project period:
1 February, 2001 - 15 June, 2001

Semester:
Dat6

Project group: E1-112a
Dennis Pedersen
Karsten Riis

Supervisor:
Torben Bach Pedersen

Copies: 6

Pages: 77

Abstract:

The changing data requirements of today's dynamic business environments are not handled well by current On-Line Analytical Processing (OLAP) systems. Physically integrating unexpected data into such systems is a long and time-consuming process making logical integration the better choice in many situations. The increasing use of Extended Markup Language (XML), e.g. in business-to-business (B2B) applications, suggests that the required data will often be available as XML data.
In this paper we present a flexible and theoretically well-founded approach to the logical federation of OLAP and XML data sources. This makes it possible to reference external XML data in OLAP queries, which allows XML data to be presented along with dimensional data in the result of an OLAP query, and enables the use of XML data for selection and grouping. Special care is taken to ensure that semantic problems do not occur in the integration process. To demonstrate the capabilities of this approach, we present a multi-schema query language based on the SQL and XPath languages. A complete federated system is also presented, covering all important areas of a federated approach to the integration of OLAP and XML. This work includes a complete formal background, a collection of algebraic rewrite rules, architectural and procedural design, and several effective cost based optimization techniques. A prototype is being developed and initial experimental studies have been conducted, indicating that our federated approach is indeed a feasible alternative to physical integration. Thus, our federated approach provides a powerful and flexible way to handle unexpected or short-term data requirements as well as rapidly changing data. As almost all data sources can be efficiently wrapped in XML format, the approach also allows the logical integration of external data from sources such as relational, object-relational, and object databases, opening up totally new application areas for OLAP.

AALBORG UNIVERSITET
INSTITUT FOR DATALOGI

Titel:

Anvendelse af XML-data i OLAP-forespørgsler

Projektperiode:
1. februar, 2001 - 15. juni, 2001

Semester:
Dat6

Projektgruppe E1-112a:
Dennis Pedersen
Karsten Riis

Vejleder:
Torben Bach Pedersen

Antal kopier: 6

Antal sider: 77

Resumé:

Mange virksomheder oplever, at deres databehov varierer over tid, hvilket er problematisk at håndtere i eksisterende systemer til On-Line Analytical Processing (OLAP). Fysisk integration af uventede data i sådanne systemer er en lang og tidskrævende proces, og logisk integration er derfor ofte en bedre løsning. Den øgede brug af Extended Markup Language (XML), for eksempel i business-to-business-applikationer (B2B), indikerer, at den slags uventede data ofte vil være tilgængelige i XML-format.
I denne artikel præsenterer vi en fleksibel og teoretisk velfunderet tilgang til logisk føderation af OLAP- og XML-datakilder. Føderationen gør det muligt at referere til eksterne XML-data i OLAP-forespørgsler, hvorved XML-data kan præsenteres sammen med multidimensionelle data i resultatet af en OLAP-forepørgsel og desuden kan bruges til selektion og gruppering. I forbindelse med integrationen sikres det, at der ikke opstår semantiske problemer. Som en demonstration af føderationstilgangen har vi konstrueret et multi-schema-forespørgselssprog baseret på sprogene SQL og XPath. Udover dette sprog præsenteres et komplet fødereret system, der dækker alle vigtige aspekter af en føderation af OLAP og XML. Dette arbejde omfatter det formelle grundlag, en samling af algebraiske omskrivningsregler, arkitekturdesign, procedurelt design samt flere effektive omkostningsbaserede optimeringsteknikker. Desuden beskrives en igangværende prototypeimplementation samt indledende eksperimenter, der indikerer, at føderation i høj grad er et praktisk alternativ til fysisk integration. Denne føderationstilgang giver således en kraftfuld og fleksibel måde at håndtere uventede og kortvarige databehov samt data, der ofte ændrer sig. Da stort set alle typer datakilder effektivt kan "indpakkes" i XML-format, tillader føderationen også logisk integration af relationelle, objekt-relationelle og objektdatabaser. Dette muliggør en række helt nye anvendelsesområder for OLAP-systemer.

Preface

This Masters Thesis presents the results of a project carried out from September 2000 to June 2001 at the Department of Computer Science, Aalborg University.

The paper describes theoretical and practical aspects of a federation of XML and OLAP databases. This work has been done in the area of *Database and Programming Technologies* during the Dat5 and Dat6 semesters. In the first semester, we contrived the fundamental definitions of data models and query languages as well as the basic federation architecture. This work constitutes most of Sections 3, 4, 5, and 7. In the second semester, we refined this foundation, and used it to create the theoretical basis for query optimization and to build and partially implement a federated system.

Citations are formed from the abbreviated name(s) of the author(s) and the year of publication, e.g. [PSGJ00]. The bibliography can be found on page 71.

We would like to thank our supervisor Torben Bach Pedersen for supervising this project, for sharing his work with us, and for his many helpful and constructive thoughts throughout the project period.

Aalborg, June 15, 2001

Dennis Pedersen

Karsten Riis

Contents

Using XML Data in OLAP Queries

Dennis Pedersen   Karsten Riis
dennisp@cs.auc.dk riis@cs.auc.dk

Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark

June 15, 2001

Abstract

The changing data requirements of today's dynamic business environments are not handled well by current On-Line Analytical Processing (OLAP) systems. Physically integrating unexpected data into such systems is a long and time-consuming process making logical integration the better choice in many situations. The increasing use of Extended Markup Language (XML), e.g. in business-to-business (B2B) applications, suggests that the required data will often be available as XML data.

In this paper we present a flexible and theoretically well-founded approach to the logical federation of OLAP and XML data sources. This makes it possible to reference external XML data in OLAP queries, which allows XML data to be presented along with dimensional data in the result of an OLAP query, and enables the use of XML data for selection and grouping. Special care is taken to ensure that semantic problems do not occur in the integration process. To demonstrate the capabilities of this approach, we present a multi-schema query language based on the SQL and XPath languages. A complete federated system is also presented, covering all important areas of a federated approach to the integration of OLAP and XML. This work includes a complete formal background, a collection of algebraic rewrite rules, architectural and procedural design, and several effective cost based optimization techniques. A prototype is being developed and initial experimental studies have been conducted, indicating that our federated approach is indeed a feasible alternative to physical integration. Thus, our federated approach provides a powerful and flexible way to handle unexpected or short-term data requirements as well as rapidly changing data. As almost all data sources can be efficiently wrapped in XML format, the approach also allows the logical integration of external data from sources such as relational, object-relational, and object databases, opening up totally new application areas for OLAP.

1 Introduction

On-line Analytical Processing (OLAP) and Extensible Markup Language (XML) are currently two of the most significant database technologies. However, the connection between them has so far received little attention.

OLAP systems enable powerful decision support based on multidimensional analysis of large amounts of summary data commonly drawn from a number of different transactional databases. OLAP data are often organized in multidimensional *cubes* containing *measured values* that are characterized by a number of hierarchical *dimensions*. Typical operations on data cubes are *roll-up*, which aggregates data by moving up along one or more dimensions, *drill-down*, which disaggregates data by moving down dimensions, and *slice-and-dice*, which performs selection and projection on a cube. The multidimensional approach offers a number of advantages over traditional types of DBMSs, including automatic aggregation [RS90], visual querying [TSC99], and good query performance due to the use of pre-aggregation [GHQ95, PJD99].

However, *dynamic data*, such as stock quotes or price lists, is not handled well in current OLAP systems, although being able to incorporate such frequently changing data in the decision making-process may sometimes be vital. Also, OLAP systems lack the necessary flexibility when faced with unanticipated or rapidly changing data *requirements*. These problems are due to the fact that physically integrating data can be a complex and time-consuming process requiring the cube to be rebuilt [Kim96, Tho97]. In some situations, the required data cannot be integrated into the cube at all e.g. because interface or copyright restrictions do not allow data to be retrieved and stored locally, but only to be queried in an ad hoc manner. Thus, logical, rather than physical, integration is desirable, i.e. a *federated* database system [SL90, BKLW99] is called for. A federated system provides a flexible way to handle rapidly changing data as well as unexpected or short term data requirements. Also, it is possible to maintain a high degree of autonomy in the component systems, e.g. these may only allow restricted access to component data. This is often the case when retrieving data from the Internet as well as in business-to-business (B2B) environments, where business partners rarely provide full access to their data. Traditionally, two different approaches have been used in the design of federations. Either component schemas are transformed into a common data model [SL90, Hsi92, DD99], or they retain their schemas and a multi-schema query language is used [PSGJ00, CRF00]. Here, the multi-schema query language approach offers the additional flexibility needed in uncertain and rapidly changing environments.

The increasing use of Extended Markup Language (XML)[W3C00], e.g. in B2B applications, suggests that the required external data will often be available in XML format. Also, most major DBMSs are now able to publish data as XML. Thus, it is desirable to be able to access XML data from an OLAP system. The hierarchical and often irregular structure of XML data allows the encoding of many different types of data, but complicates its use in connection with more structured types of data. For example, irregular XML data can lead to incorrect aggregation of data if not handled properly.

In this paper we present a theoretically well-founded approach to the logical federation of OLAP and XML data sources. The approach allows external XML data to be used as “virtual” dimensions, enabling three specific uses of XML data. First, OLAP query results may be “*decorated*” with XML data, i.e. related XML data may be presented along with the results of an OLAP query. Second, external XML data may be used for *selection*. Third, OLAP data may be *grouped* based on external XML data. Special care is taken to ensure that any irregularities in the XML data does not cause problems w.r.t. correct aggregation of data. A flexible linking mechanism is devised to associate dimensional cube data with parts of XML documents, allowing XML data to be referenced in a multidimensional query. As almost all data sources can be efficiently wrapped in XML format [CCS00], the presented approach also allows external data from sources such as relational, object-relational, and object databases to be used in a powerful and flexible way. This extends the use of OLAP to completely new application areas as data need no longer be integrated physically in the OLAP DB.

The work presented here covers all important areas of a federated approach to the integration of OLAP and XML, including a complete formal background, architectural and procedural design, several cost based optimization techniques, a prototype implementation, and experimental studies.

To demonstrate the capabilities of the approach, we present a data model, a formal algebra over the model and a high-level multi-schema query language based on SQL and XPath [W3C99]. SQL and XPath are chosen for their simplicity, wide-spread use, and compact syntax. However, other languages like MDX [Mic98] or XQuery [W3C01b] could be used instead. The extended SQL is called *XML-Extended Multidimensional SQL* (SQL_{XM}). As a part of this work, a new data model and query language has been defined for OLAP. To demonstrate the federation, a data model and a query language should satisfy two requirements: First, they should be capable of handling the irregularities introduced by integration with external data. Second, they should be close to the relational model and SQL to facilitate the integration with existing technology and to ease user comprehension. Unfortunately, we found the existing data models either too simple for the first requirement or too complex for the second requirement [PJ99]. Thus, a new data model and query language has been defined. This query language is called SQL_M.

We also present an overview of a prototype system supporting the approach, which is currently in the early stages of implementation. In addition, a number of effective optimization techniques are described. Together, they provide significantly better performance for typical queries and also allow queries to be evaluated, that would otherwise be orders of magnitude too expensive. For example, a technique is presented to *inline* references to external data in queries, potentially resulting in much faster evaluation. Another primary result used for optimization, is the definition of a set of *equivalence rules* for the federation algebra. Optimizations are based on a *cost model* for federation queries, and techniques are presented that allow cost estimates to be made even when little or no cost information is available. Without loss of generality, we make no assumptions about the existence of Document Type Definitions [W3C00] or XML Schemas [W3C01a] which ensures compatibility with all kinds of XML sources. Initial performance experiments with the prototype are reported, indicating that our approach is indeed a practical alternative to physical integration. The concepts of the approach are illustrated by a real-world case study based on the use of OLAP and XML systems in the B2B domain.

There has been a great deal of previous work on data integration, e.g., on integrating relational data [HSC99, Dat01, Gat01], object-oriented data [RAH$^+$96], semi-structured data [CGMH$^+$94], and a combination of relational and semi-structured data [GW00, LAW99]. However, none of these handle the advanced issues related to OLAP systems, e.g., dimensions with hierarchies, automatic aggregation, and the problems related to correct aggregation. This is also true for the combined relational and XML query language Quilt [CRF00], and for nD-SQL [GL98], which considers the federation of relational sources providing basic OLAP functionality. One previous paper [PSGJ00] has considered the federation of OLAP and object data. In comparison, our approach is not restricted to object DBs, and their rigid schemas, but can be used on any imaginable data source as long as it allows XML wrapping. Also, we allow irregularities in federation data such as "missing" external data and offer more general use of external data when performing selection and grouping. The same paper briefly mentions the *inlining* technique, but only for certain simple types of predicates. Also, we consider a cost based use of the technique.

We believe this paper to be the first to consider the logical integration of OLAP and XML data, opening up totally new application areas for OLAP as physical integration of data is no longer needed. More specifically, we believe to be the first to:

- define a data model, an algebra, and a query language for OLAP which support irregular hierarchies and are close to the relational model and SQL,
- define a data model, an algebra, and a multischema query language for the federation of OLAP and XML data sources,
- consider advanced OLAP issues such as dimension hierarchies, automatic aggregation, and correct aggregation of data in the context of integration with XML data,
- formally define the *decoration operation* as a basis for logical integration of external data in OLAP systems,
- demonstrate how the decoration operation enables selection and grouping based on external data,
- formulate equivalence rules involving the decoration operation,
- propose a cost model for federation queries based on specific cost models for autonomous OLAP and XML components,
- devise a number of effective heuristic and cost based optimization techniques for the federation queries,
- present a general form of the *inlining* technique for integration of external data in predicates.

The rest of the paper is organized as follows. Section 2 presents the motivation and the case study used throughout the paper, while Section 3 defines the data models and query languages used in the federation components. In Section 4, the linking mechanism and its use in OLAP-XML federations is defined, while Section 5 defines the semantics of the approach. Equivalence rules for the federation algebra are presented in Section 6. Section 7 and Section 8 describes the federation architecture and how queries are evaluated within the architecture, respectively. A number of optimization techniques and a high level cost model are presented in Section 9 followed by a more detailed explanation of the cost model in Section 10. An overview of the federated system is presented in Section 11. Finally, Section 12 discusses implementation and performance studies, whereas Section 13 concludes the paper and points to future work. Appendices A and B contain the formal syntax of the SQL_{XM} language and a formal definition of the inlining approach, respectively.

2 Motivation

In this section, we discuss why a federation of existing OLAP and XML databases may often be useful and present a real-world case study that is used for illustration throughout the paper.

2.1 Federating OLAP and XML

As described in the introduction, this work is aimed at, but not limited to, the use of XML data from autonomous sources, such as the Internet, in conjunction with existing OLAP systems. Our solution is to make a federation which allows users to quickly define their own *logical cube view* by creating *links* between existing dimensions and XML data. This immediately permits queries that use these new "virtual" dimensions in much the same way ordinary dimensions can be used. For example, in a cube containing data about sales, a Store-City-Country dimension may be linked to a public XML document with information about cities, such as state and population. Instead of being restricted to queries that use only the existing dimensions, like "Show sales by month and city", it is now possible to pose queries such as "Show sales by month and state" or "Show sales by month and city population". Thus, in effect the cube data can be *grouped by* XML data residing e.g. on a web page or in a database with an XML interface. In addition, such data can be used to perform *selection* (also known as filtering) on the cube data, e.g. "Show only sales for cities with a population of more than 100.000" or to *decorate* dimensions, e.g. "Show sales by month and city and for each city, show also the state in which it is located".

Many types of OLAP systems may benefit from being able to logically integrate external XML data. In a business setting, consider e.g. an OLAP database containing data about products and their production prices. To aid in determining future sales prices, these products could be decorated with a competing company's prices for the same or similar products. Such prices would typically be available from the competing company's website. Another example could be various types of geographical information available from the Web. These may be used in an insurance company's customer database, a hospital's admissions database, or a telephone company's database of calls. Other types of external data that will typically be available as XML data and could be useful in OLAP systems include: addresses of employees or customers, product specifications, the store manager's name, dates of events, public assessment for the taxes on real estate etc. Thus, a broad range of different systems can gain from our federation.

This federated approach where *users* are responsible for defining the federation, has been referred to as a *loosely coupled federation* [SL90]. There are many reasons why this approach is a good choice for this setting. It provides the ability to do *ad hoc integration*, which may be needed for a number of reasons. First, it is rarely possible to anticipate all future data requirements when designing a database schema. OLAP databases may contain large amounts of data and thus, physically integrating the data can be a time consuming process requiring a partial or total rebuild of the cube. However, being able to quickly obtain the necessary data can sometimes be vital in making the right strategic decision. Second, not all types of data are feasible to copy and store locally even though it is available for browsing e.g. on the Internet. Copying may not be allowed, typically because of copyright rules, or it may not be practical, e.g. because data changes too frequently or because only limited form-based interfaces are available, requiring each data item to be explicitly requested. Third, attempting to anticipate a broad range of future data needs and physically integrating the data increases the complexity of the system, thereby reducing maintainability. Also, this may degrade the general performance of the system. Finally, ad hoc integration allows *rapid prototyping* of OLAP systems, which can significantly ease the task of deciding which data to physically integrate. Altogether, the federated approach provides both a *simple* and a *special purpose* OLAP system as data can be integrated when the need arises.

The federated approach also allows XML components to maintain a *high degree of autonomy*, which is essential when data is accessed from sources outside the organisation that controls the federation. For example, when a component is accessed on the Internet, the federation will typically have no control over the component's structure, naming conventions, access methods, availability etc. Also, data is always *up-to-date* when using a federated system as opposed to physically integrating the data. This may be crucial for certain types of dynamic data such as price lists, stock quotes, contact information, scheduled dates etc.

These points suggest that in many situations a loosely coupled federation is preferable.

2.2 Case study

The case study concerns the trading of electronic components. It is inspired by the Electronic Component Information Exchange (ECIX)[Eci01], which is a widely adopted initiative to use XML as a means of communicating information about electronic components. The setting, simplified to fit this paper, consists of companies producing electronic components (ECs), and of companies buying these components and integrating them to larger appliances. In the following we refer to them as suppliers and customers, respectively.

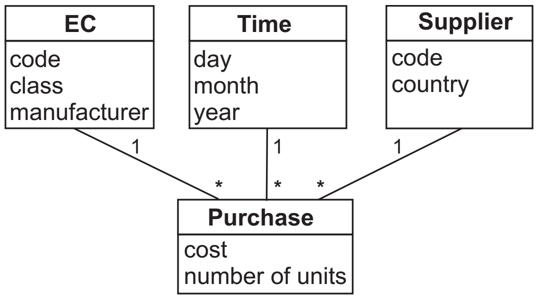


Figure 1: UML schema for the Purchases OLAP database.

Customers use an OLAP database, shown in the UML diagram in Figure 1, to analyze the purchases they have made over time. Purchases are characterized by an EC dimension, a supplier dimension, and a time dimension, and for each purchase the total cost and the purchased amount are measured. ECs are categorized by their manufacturers and their classes, e.g. flip-flops or latches. For suppliers, we capture the country in which they are located. Purchase dates are categorized according to the regular calendar. This database allows customers to view purchases at different levels of granularity e.g. to calculate the total amount spent on ECs by class and month.

Suppliers present their products on the Web at a B2B marketplace. This allows customers and others to access detailed specifications of their ECs. This information is encoded in an industry-wide markup language defined in XML, which makes it easy to limit a search to the relevant parts of specifications. A simplified example of a document containing information from different suppliers is shown in Figure 2. The fundamental part of an XML document is the *element*. Elements are identified by a *start tag* and an *end tag*, and can contain other elements, text data, and attributes. For a more comprehensive explanation of XML see [W3C00]. In the example document the `Component` element has an attribute `CompCode` and contains the elements `Manufacturer`, `UnitPrice` and `Description`. All ECs sold by a particular supplier belong to a component class. ECs are referred to by their code. In addition to this, a document captures the manufacturer, which need not be the same as the supplier, the price per unit, and a textual description.

Several aspects of ECs like textual descriptions and current prices are not included in the Purchases database because their use was not anticipated or because they change too frequently. Despite this, it may sometimes be desirable e.g. to group ECs by their marketplace descriptions, or view only purchases of ECs within a specific price range. By logically integrating the Purchases database and the Components document in a federation this can be handled in an easy and flexible way.

```
<?xml version="1.0" encoding="utf-8"?>
<Components>
  <Supplier SCode="SU13"><SName>John’s ECs</SName>
    <Class ClassCode="C24"><ClassName>Flip-flop</ClassName>
      <Component CompCode="EC1234">
        <Manufacturer MCode="M31"><MName>Smith Components Inc.</MName></Manufacturer>
        <UnitPrice Currency="euro" NoOfUnits="1000">3.00</UnitPrice>
        <UnitPrice Currency="euro" NoOfUnits="10000">2.60</UnitPrice>
        <Description>16-bit flip-flop</Description>
      </Component>
      <Component CompCode="EC1235">
        <Manufacturer MCode="M32"><MName>John’s ECs</MName></Manufacturer>
        <UnitPrice Currency="euro" NoOfUnits="1000">4.25</UnitPrice>
        <Description>16-bit flip-flop</Description>
      </Component>
    </Class>
  </Supplier>
  <Supplier SCode="SU15"><SName>Jane’s ECs</SName>
    <Class ClassCode="C27"><ClassName>Latch</ClassName>
      <Component CompCode="EC2346">
        <Manufacturer MCode="M31"><MName>Smith Components</MName></Manufacturer>
        <UnitPrice Currency="euro" NoOfUnits="1000">3.31</UnitPrice>
        <Description>16-bit latch</Description>
      </Component>
    </Class>
    <Class ClassCode="C24"><ClassName>Flip-Flop</ClassName>
      <Component CompCode="EC1234">
        <Manufacturer MCode="M33"><MName>Johnson Components</MName></Manufacturer>
        <UnitPrice Currency="euro" NoOfUnits="1000">2.95</UnitPrice>
        <Description>D-type flip-flop</Description>
      </Component>
    </Class>
  </Supplier>
</Components>
```

Figure 2: The Components document containing information about EC suppliers and their products.

3 Data Models and Query Languages

This section describes the data models and query languages used for the federated components. For the OLAP component, a prototypical model capturing common multidimensional terms such as facts, dimensions, and hierarchies is defined, and an OLAP-extended version of SQL is used as the query language. The OLAP data model captures complex multidimensional data, e.g., irregular dimension hierarchies, just as the model of Pedersen et al. [PJ99], but has been modified to be closer to standard SQL. For a description of how to implement such an advanced model using standard OLAP technology, we refer to previous work [PJD99]. For the XML component, the XPath data model and query language [W3C99] is used, mainly because of its simplicity and wide-spread use.

3.1 The OLAP Data Model

The model is defined in terms of a multidimensional *cube* consisting of a *cube name*, *dimensions*, and a *fact table*. Each dimension comprises two partially ordered sets (posets) representing hierarchies of *levels* and the ordering of *dimension values*. Each level is associated with a set of dimension values.

Definition 3.1 (Dimension) A *dimension* D_i is a two-tuple (L_{D_i}, E_{D_i}), where L_{D_i} is a poset of levels and E_{D_i} is a poset of dimension values.

L_{D_i} is the four-tuple $(LS_i, \sqsubseteq_i, \top_i, \bot_i)$, where $LS_i = \{L_{i1}, \ldots, L_{ik_i}\}$ is a set of levels, $\sqsubseteq_i$ is a partial order on these levels, and $\top_i$ and $\bot_i$ are the unique top and bottom elements of the ordering. We shall use $L_{ij} \in D_i$ as a shorthand meaning that the level L_{ij} belongs to the poset of levels in dimension D_i.

A *level* L_{ij} is a name identifying a set of *dimension values*. Let E be the set of all possible dimension values and $Levels$ be the set of all levels. Then a function $\text{Values} : Levels \mapsto \mathcal{P}(E)$, returns the subset of E associated with a level in $Levels$. Thus, $\text{Values}(L_{ij}) = \{e_{ij1}, \ldots, e_{ijl_{ij}}\}$. We shall use L_{ij} as a shorthand for $\text{Values}(L_{ij})$.

E_{D_i} is a poset $\left(\bigcup_j L_{ij}, \sqsubseteq_{D_i}\right)$, consisting of the set of all dimension values in the dimension and a partial ordering defined on these. We shall use D_i as a shorthand for $\bigcup_j L_{ij}$.

For each level L we assume a function $\text{Roll-up}_L : \text{Values}(L) \times LS_i \mapsto \mathcal{P}(D_i)$, which given a dimension value in L and a level in LS_i returns the value's ancestors in the level. That is, $\text{Roll-up}_L(e, L') = \{e' \in L' | e \sqsubseteq_{D_i} e'\}$. □

The intuition behind the partial order $\sqsubseteq_i$ of levels is that given two levels $L_{i1}, L_{i2} \in D_i$ we say that $L_{i1} \sqsubseteq_i L_{i2}$ if elements in L_{i2} can be said to contain the elements in L_{i1}. For example, $Day \sqsubseteq Year$ because years contain days. Similarly, we say that $e_1 \sqsubseteq e_2$ if e_1 is logically contained in e_2 and $L_{ij} \sqsubseteq_i L_{ik}$ for $e_1 \in L_{ij}$ and $e_2 \in L_{ik}$ and $e_1 \neq e_2$. For example, the day 01.21.2000 is contained in the year 2000. Note that a lower-level value may roll up to more than one higher-level value.

Example 3.1 In the case study presented in Section 2.2 we have a Time dimension, an ECs dimension and a Suppliers dimension. Letting Sup denote $Suppliers$ the Suppliers dimension consists of the levels $LS_{Sup} = \{\top_{Sup}, Country, Supplier\}$, which are ordered as follows: $\sqsubseteq_{Sup} = \{(Supplier, \top_{Sup}), (Country, \top_{Sup}), (Supplier, Country)\}$. Thus, the poset of levels is $L_{D_{Sup}} = (LS_{Sup}, \sqsubseteq_{Sup}, \top_{Sup}, Supplier)$.

The poset of dimension values are $E_{D_{Sup}} = (\{\top_{D_{Sup}}, US, UK, S1, S2, S3\}, \sqsubseteq_{D_{Sup}})$, where $\sqsubseteq_{D_{Sup}} = \{(US, \top_{D_{Sup}}), (UK, \top_{D_{Sup}}), (S1, \top_{D_{Sup}}), (S2, \top_{D_{Sup}}), (S3, \top_{D_{Sup}}), (S1, US), (S2, US), (S3, UK)\}$.

Hence, the Suppliers dimension is given by: $D_{Sup} = (L_{D_{Sup}}, E_{D_{Sup}})$. □

Definition 3.2 (Fact table) A *fact table* F is a relation containing one attribute for each dimension and one attribute for each measure. Thus, $F = \{(e_{\perp_1}, \ldots, e_{\perp_n}, v_1, \ldots, v_m) | \ (e_{\perp_1}, \ldots, e_{\perp_n}) \in \perp_1 \times \cdots \times \perp_n \wedge (v_1, \ldots, v_m) \in M \subseteq T_1 \times \cdots \times T_m\}$, where $n \geq 1$, $m \geq 1$, and T_j is the domain value for the j'th measure. We will also refer to the j'th measure as $M_j = \{(e_{\perp_1}, \ldots, e_{\perp_n}, v_j)\}$. The measure domains T_j all contain the special NULL value, which denotes that no value exists for a particular combination of dimension values. A tuple in F, where at least one measure value exists, is called a *fact*.

Each measure M_j is associated with a *default aggregate function* $f_j : \mathcal{P}(T_j) \mapsto T_j$, where the input is a multi-set. Aggregate functions ignore NULL values as in SQL. □

Intuitively, a tuple in F captures the measured values associated with one combination of dimension values from the bottom levels. The number of tuples in F is equal to $\|\perp_1\| \cdot \cdots \cdot \|\perp_n\|$. That is, there is one tuple in F for each possible combination of the bottom dimension values. This is just logically, in a physical implementation only the non-empty tuples would be stored.

Example 3.2 In the case study presented in Section 2.2 we have the two measures *Cost* and *Number of Units*. A part of the fact table is represented in Table 1. To save space, only tuples with non-NULL measure values are shown although all combinations are logically present in the relation. This is done throughout the paper. □

Definition 3.3 (Cube) An n*-dimensional cube* C is a three-tuple consisting of a cube name N, a non-empty set of dimensions $D = \{D_1, \ldots, D_n\}$ and a fact table $F(D_1, \ldots, D_n, M_1, \ldots, M_m)$. That is: $C = (N, D, F)$. The *cube name* N describes the type of facts contained in the cube. □

Example 3.3 From the Purchases database in the case study we can construct a three-dimensional cube with the cube name $Purchases$, the dimensions, levels, and ordering of dimension values as depicted in Figure 3, and the fact table from Example 3.2. □

Cost	No. Of Units	Day	Supplier	EC
2940	1000	01.21.2000	S1	EC1234
6900	2000	01.21.2000	S3	EC1234
9480	3000	02.22.2000	S3	EC2345
14400	4000	02.22.2000	S2	EC1235
17650	5000	03.23.2001	S2	EC1235

Table 1: A part of the fact table for the Purchases database.

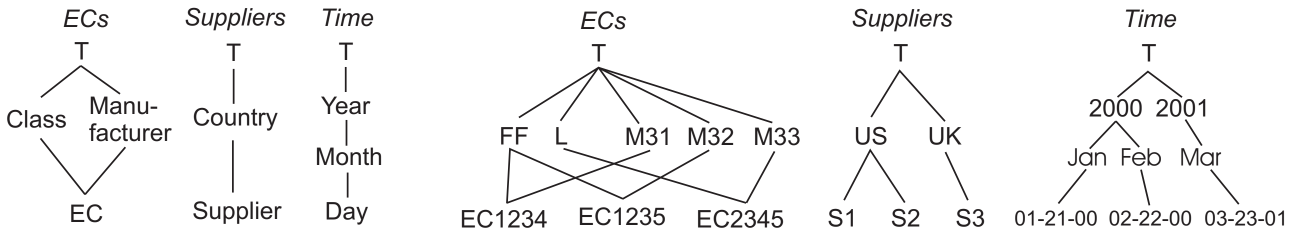


Figure 3: Schema (left) and instance (right) of the Purchases database. "FF" and "L" are names of classes denoting "Flip-flops" and "Latches", respectively.

Next, we define the notion of *summarizability* and discuss how it is used to ensure *safe aggregation*. Summarizability is an important cube property as it states when lower-level aggregates, which are often pre-computed, can be used to calculate higher-level aggregates, and when these must be computed from base data. Also, it is possible to get wrong results from aggregate queries if summarizability is not ensured. Checking for summarizability is even more important in the setting considered in this paper than in normal OLAP systems, as the irregular structure of XML data may violate the summarizability property.

Definition 3.4 (Summarizable) Given a measure domain T_M, a set $S = \{S_1, \ldots, S_k\}$ where $S_j \in \mathcal{P}(T_M)$, and a function $g : \mathcal{P}(T_M) \mapsto T_M$ we say that g is *summarizable* if $g(\{g(S_1), \ldots, g(S_k)\}) = g(S_1 \cup \cdots \cup S_k)$ where the argument of the left-hand side is a multiset. □

Intuitively, an aggregate function is summarizable if aggregated results from a lower-level aggregate (left-hand side of the formula) can be combined to give the same result as when the aggregate is derived directly from base data (right-hand side of the formula). If this property is not satisfied we are generally not allowed to use the lower-level results for further aggregation.

It has been shown that summarizability is equivalent to requiring the aggregate function to be distributive, and the ordering of dimension values to be *strict*, *onto*, and *covering* [LS97, PJ99]. A hierarchy is *strict* if no dimension value has more than one parent value from the same level, *onto* if all paths from top value to leaf value is of equal length, and *covering* if no path skips one or more levels. Intuitively, this means that dimension hierarchies must be balanced trees. If this is not the case some lower-level values will be either double-counted or not counted at all. For example, the Purchases cube in Figure 3 is strict, onto, and covering.

Another important problem is that different aggregate functions may be valid when aggregating different measures and also when aggregating the same measure over different dimensions. Consider e.g. the two measures "number of items sold per store per year" and "number of items in stock per store per year". The first one can be added meaningfully over time, e.g. to "number of items sold per store per decade". However, the second one is not summarizable over the time dimension as some item may be accounted for more than once. The average number of items would be meaningful, though. To compute the stock items per decade, it is necessary to use the base data, i.e. each stock item. This purely semantical difference between the two types of measures is often characterized using the terms *flow* and *stock*, respectively [LS97]. Even though stock items cannot be added over the time dimension, it may be correct to aggregate them over the store dimension, e.g. by country.

Notice that this is only correct if items are only stored in one place.

To ensure correct aggregation of data we keep track of which aggregate functions can meaningfully be applied to measures for each dimension. We do this by associating an *aggregation type* to each combination of a measure and a dimension, thereby allowing us to prohibit or warn the user against illegal aggregations. Following previous work [RR83, Leh98, PJ99], we distinguish between three types of data: c, data that may not be aggregated because summarizability is not preserved, ϕ, data that may be averaged but not added, and Σ, data that may also be added. Thus, we have the following ordering of these types: $c \subset \phi \subset \Sigma$. Considering only the standard SQL functions, we have that $\Sigma = \{\text{SUM}, \text{AVG}, \text{MAX}, \text{MIN}, \text{COUNT}\}$, $\phi = \{\text{AVG}, \text{MAX}, \text{MIN}, \text{COUNT}\}$, and $c = \emptyset$. A function $\text{AggType} : \{M_1, \ldots, M_m\} \times D \mapsto \{\Sigma, \phi, c\}$ returns the aggregation type of a measure M_j when aggregated in a dimension $D_i \in D$. Thus, any changes to an aggregation type apply to all levels in a dimension. Aggregation types are used both to prohibit semantically incorrect aggregation, and to prevent aggregation when irregular hierarchies may lead to incorrect results. We only consider the problem of non-strict hierarchies, since this is the only type of irregularity introduced when integrating external data as discussed in Section 5. Hence, we assume that cubes only contain hierarchies that are onto and covering.

3.2 The Cube Algebra

In this section we present an algebra over the OLAP data model presented in Section 3.1. Two operators are defined: a selection operator and a generalized projection operator. Hence, the algebra is not relationally complete, but it is sufficiently powerful for this purpose. For examples of a complete set of operators, we refer to previous work [PJ99].

The selection operator σ_{Cube} is used to slice the cube so that it contains only facts that fulfill a given predicate. The predicates we consider here are constructed from the usual SQL operators, and allow the use of roll-up functions on the form $L'(L)$ which returns the dimension values in L' that contain each dimension value in L. Assuming unique level names, this can be abbreviated to L'.

Example 3.4 Slice the purchase cube from Example 3.3 so that only the measured values for ECs not supplied by ‘S2’ and belonging to classes starting with an ‘F’ are retained in the cube:

$\sigma_{Cube[\text{Supplier<>‘S2’ AND Class LIKE ‘F%’}]}(Purchases) = Purchases'$ □

A predicate may have more than one interpretation if a dimension value can have more than one parent in the same level, i.e. if the hierarchy is non-strict. This would be the case when selecting ECs belonging to classes that start with an “F”. The predicate could be true if *all* classes to which an EC belongs begins with an “F” or it could be true if *any* of the classes do so. We call these semantics *all selection semantics* and *any selection semantics*, respectively. Here, we adopt the latter interpretation, since we consider this the more natural choice for users and since it is also the one used in the XPath standard [W3C99]. A selection only affects the tuples in the fact table. Hence, selection returns a cube with the same fact type and the same set of dimensions. All tuples for which the predicate holds are left unaffected. For all other tuples the measures are set to NULL.

Example 3.5 The fact table resulting from the query in Example 3.4 is:

Cost	No. Of Units	Day	Supplier	EC
2940	1000	01.21.2000	S1	EC1234
6900	2000	01.21.2000	S3	EC1234

□

Formally, we define the selection operator as follows:

Definition 3.5 (Selection operator) Let p be a predicate over the set of levels $\{L_1, \ldots, L_k\}$ and measures $M_1, \ldots, M_m$. Selection on a cube $C = (N, D, F)$ is $\sigma_{Cube[p]}(C) = (N', D', F')$, where $N' = N$, $D' = D$

Cost	Supplier	Class
2940	S1	FF
6900	S3	FF
9480	S3	L
32050	S2	FF

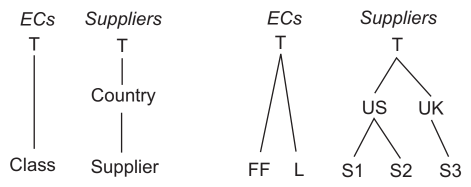


Figure 4: The $Purchases'$ cube from Example 3.6

and $F' = \{t'_1, \ldots, t'_l\}$. If $t_i = (e_{\perp_1}, \ldots, e_{\perp_n}, v_1, \ldots, v_m) \in F$ then

$$t'_i = \begin{cases} t_i & \text{if } p(t_i) = tt \\ (e_{\perp_1}, \ldots, e_{\perp_n}, \text{NULL}, \ldots, \text{NULL}) & \text{otherwise.} \end{cases}$$

□

The generalized projection operator Π_{Cube} aggregates measures to a given level and at the same time removes dimensions and measures from a cube. This is similar to the behavior of a SELECT statement with a GROUP BY clause in SQL.

Generalized projection is evaluated in three steps: First, we remove all dimensions that are not present in the arguments, and then each dimension value is rolled up to the specified level. Finally, we perform a regular grouping in the fact table removing all measures not specified in the arguments. Notice that rolling up to a higher level may result in duplicated facts if the hierarchy is non-strict.

Example 3.6 Calculate the costs per class and supplier:

$$\Pi_{Cube[Supplier, Class] < \text{SUM}(Cost)>}(Purchases) = Purchases'$$

The $Purchases'$ cube resulting from this query is shown in Figure 4. □

Intuitively, the levels specified as an argument to the operator becomes the new bottom levels of their dimensions and all other dimensions are aggregated to the top level and removed. Each new measure value is calculated by applying the given aggregate function to the corresponding value for all tuples in the fact table containing old bottom values that roll up to the new bottom values. To ensure safe aggregation in case of non-strict hierarchies we explicitly check for this in each dimension. If a roll-up along some dimension duplicates facts we disallow further aggregation along that dimension by setting the aggregation type to c.

Formally, we define:

Definition 3.6 (Generalized projection) Let $C = (N, D, F)$ be a cube as defined above. Then generalized projection is defined as: $\Pi_{Cube[\mathcal{L}_{i_1}, \ldots, \mathcal{L}_{i_k}] < f_{j_1}(M_{j_1}), \ldots, f_{j_l}(M_{j_l})>}(C) = (N', D', F')$, where $\{\mathcal{L}_{i_1}, \ldots, \mathcal{L}_{i_k}\}$ is a set of levels specifying the aggregation level such that at most one level from each dimension occurs. The measures $\{M_{j_1}, \ldots, M_{j_l}\} \subseteq \{M_1, \ldots, M_m\}$ are kept in the cube and $f_{j_1}, \ldots, f_{j_l}$ are the given aggregate functions for the specified measures, such that $\forall D'_g \in \{D_g \in D | \perp_g \notin \{\mathcal{L}_{i_1}, \ldots, \mathcal{L}_{i_k}\}\}(\forall f_{j_h} \in \{f_{j_1}, \ldots, f_{j_l}\}(f_{j_h} \in \text{AggType}(M_{j_h}, D'_g)))$.

The resulting cube is given by: $N' = N$ and $D' = \{D'_{i_1}, \ldots, D'_{i_k}\}$, where $D'_{i_h} = (L'_{D_{i_h}}, E'_{D_{i_h}})$ for $h = 1, \ldots, k$. The new poset of levels in the remaining dimensions is $L'_{D_{i_h}} = (LS'_{i_h}, \sqsubseteq'_{i_h}, \top_{i_h}, \mathcal{L}_{i_h})$, where $LS'_{i_h} = \{L_{i_h p} \in LS_{i_h} | \mathcal{L}_{i_h} \sqsubseteq_{i_h} L_{i_h p}\}$, and $\sqsubseteq'_{i_h} = \sqsubseteq_{i_h|_{LS'_{i_h}}}$. Moreover, $E'_{D_{i_h}} = (\bigcup_p L_{i_h p}, \sqsubseteq_{D_{i_h}|\bigcup_p L_{i_h p}})$.

The new fact table is given by: $F' = \{(e'_{\perp_{i_1}}, \ldots, e'_{\perp_{i_k}}, v'_{j_1}, \ldots, v'_{j_l}) | \; e'_{\perp_{i_g}} \in \mathcal{L}_{i_g} \wedge v_{j_h} = f_{M_{j_h}}(\{v | (e_{\perp_1}, \ldots, e_{\perp_n}, v) \in M_{j_h} \wedge (e'_{\perp_{i_1}}, \ldots, e'_{\perp_{i_k}}) \in \text{Roll-up}_{\perp_{i_1}}(e_{\perp_{i_1}}, \mathcal{L}_{i_1}) \times \cdots \times \text{Roll-up}_{\perp_{i_k}}(e_{\perp_{i_k}}, \mathcal{L}_{i_k})\})\}$.

Furthermore, if $\exists (e_{\perp_1}, \ldots, e_{\perp_n}, v_j) \in M_{j_h}(\exists e \in \{e_{\perp_1}, \ldots, e_{\perp_n}\}(\|\text{Roll-up}_{\perp_{i_g}}(e, \mathcal{L}_{i_g})\| > 1 \wedge v_j \neq \text{NULL}))$ then $AggType(M_{j_h}, D'_{i_g}) = c$. □

3.3 The OLAP Query Language

We use a slightly extended subset of SQL, called Multidimensional SQL (SQL_M), to query multidimensional cubes. SQL is chosen as the base language because of its simplicity and wide-spread use. We illustrate the considered syntax with an example, while the full syntax is given in Appendix A.

Example 3.7 Calculate costs by class and supplier but only for suppliers located in UK and only when the total cost exceeds 10000:

```
SELECT     SUM(Cost), Supplier, Class(EC)
FROM       Purchases
WHERE      Country(Supplier) = 'UK'
GROUP BY   Supplier, Class(EC)
HAVING     SUM(Cost) > 10000
```

□

A query is constructed from SQL's SELECT-FROM-WHERE-GROUP BY-HAVING statement, with a few modifications to the standard language to capture the special OLAP concepts. Aggregation from a bottom level L to a higher level L' in the same dimension is performed using a roll-up function $L'(L)$ in the SELECT and GROUP BY clauses. Like [AGS97], we assume these functions to be multi-valued although this is not possible in standard SQL. This is necessary because we allow hierarchies to be non-strict, e.g. ECs could belong to more than one class. Roll-up functions can also be used in the WHERE and HAVING clauses. Since the dimension to which the levels belong is not given in the syntax, we assume level names to be unique. This can be handled by prepending level names with dimension names. As a shorthand, we allow the argument of the roll-up function to be omitted if $L' = L$, that is when no roll-up should be performed in that particular dimension.

Since we do not allow relational projection on cubes the GROUP BY clause is mandatory. Each dimension must either be explicitly rolled up to some level or not mentioned at all. The latter indicates that the dimension should be rolled up to the top level and projected away as is the case in standard SQL. However, if only measures are to be removed from the cube, that is if all bottom levels are present in the SELECT clause, and no HAVING clause is present, the GROUP BY clause can be omitted.

To support automatic aggregation a new function DEFAULT can be used in addition to the usual SQL aggregate functions. When applied to a measure M_j "DEFAULT" is substituted for the default aggregate function f_j. For example, if $f_{\text{Cost}} = \text{SUM}$ then DEFAULT(Cost) becomes SUM(Cost). In this way a user need not be concerned with the aggregate functions once they are specified in the system. This is important, e.g. when OLAP data is queried using a graphical tool.

Without loss of generality we only allow one cube in the FROM clause and we do not consider calculated measures. Both multiple cubes and calculated measures can be handled by creating a view over one or more cubes. We do allow nested queries in the FROM clause.

The semantics of an SQL query can now be expressed in terms of the cube algebra defined above: First, the selection operator is applied with the predicate from the WHERE clause, then generalized projection with the levels and measures listed in the SELECT and GROUP BY clauses, and finally a new selection is performed with the HAVING predicate.

Example 3.8 The query in Example 3.7 is evaluated as follows:

$$Purchases' = \sigma_{Cube[\text{SUM}(Cost)>10.000]}(\Pi_{Cube[Supplier, Class]<\text{SUM}(Cost)>}(\sigma_{Cube[Country='UK']}(Purchases)))$$

□

Formally, we define the semantics of an SQL query as follows:

Definition 3.7 (Semantics of an SQL-query) Let C be a cube, $\{L_1, \ldots, L_k\}$, and $\{L'_1, \ldots, L'_k\}$ be a set of levels from a subset of dimensions in C where $L_i \sqsubseteq_i L'_i$, $\{M_1, \ldots, M_l\}$ be a set of measures from C, $pred_{where}$ be a predicate over the levels and measures in C, and $pred_{having}$ be a predicate over levels $L'_1, \ldots, L'_k$, and measures $M_1, \ldots, M_l$.

The SQL_M-query

SELECT	$f_1(M_1), \ldots, f_l(M_l), L'_1(L_1), \ldots, L'_k(L_k)$
FROM	C
WHERE	$pred_{where}$
GROUP BY	$L'_1(L_1), \ldots, L'_k(L_k)$
HAVING	$pred_{having}$

can then be evaluated as $C' = \sigma_{Cube[pred_{having}]}(\Pi_{Cube[L'_1, \ldots, L'_k] < f_1(M_1), \ldots, f_l(M_l) >}(\sigma_{Cube[pred_{where}]}(C)))$. □

3.4 The XML Data Model and Query Language

The XPath language is used to refer to parts of XML documents. Although not a full blown query language, this language is sufficiently powerful for our purpose. XPath is also chosen because it has a compact syntax making it suitable for integration into another language. Given this language, the natural choice for an XML data model is the tree based model underlying the XPath language. Space constraints prohibit a complete definition of the language and the reader is referred to the XPath specification for details [W3C99].

The XML data model underlying the XPath language views an XML document as a tree. Each node in the tree has one of the types: root, element, namespace, text, processing instruction, attribute, or comment. The types are defined by the set $V = \{R, E, N, T, P, A, C\}$.

The tree structure appears because some of these nodes can contain other nodes, whereas others contain just primitive parts like text or strings. The contents of the nodes are defined in the following.

In the definition $r, e, n, t, p, a,$ and c are nodes of type $R, E, N, T, P, A,$ and C, respectively. An expanded name is composed of a local name and possibly a namespace part making the name globally unique. CDATA stands for character data, i.e. a piece of text.

$$r = (e, \{p_1, \ldots, p_{k_1}\}, \{c_1, \ldots, c_{k_2}\})$$
$$e = (Expanded\,name, Id, (e'_1, \ldots, e'_{k_3}), \{n_1, \ldots, n_{k_4}\}, \{p_1, \ldots, p_{k_5}\}, \{a_1, \ldots, a_{k_6}\}, \{c_1, \ldots, c_{k_7}\})$$

where e'_i is either an element or a text node, and the order of these nodes is given by the order in which start tags occur in the document. Note that an element need not have an id, in which case the id has the special value NULL.

$$\begin{aligned} n &= (Expanded\,name) \\ t &= (CDATA) \\ p &= (Expanded\,name, String) \\ a &= (Expanded\,name, String) \\ c &= (String) \end{aligned}$$

We define an XML document x as a pair $x = (URI, r)$, where URI is the globally unique name of the XML document and Type$(r) = R$. We further assume a function Root$(x) =$ Root$((URI, r)) = r$ that returns the root of an XML document x, a function StrVal(s) that returns a string representation of a node s, and a function Nodes(x) that returns the set of nodes in an XML document x.

The basic syntax of an XPath expression resembles a Unix file path where a full path expression is given as a number of locations separated by a "/", e.g. location-step$_1$/…/location-step$_n$. Each step in turn selects a set of nodes relative to a *context node*. Each node in that set is then used as a context node for the next step. The syntax for one step is: axis::node-test[predicate$_1$]...[predicate$_n$]

The axis part of a location step specifies the tree relationship between the nodes selected by the location step and the context node. These include attribute, parent, and child which can be abbreviated by "@", ".." and by omitting the axis, respectively. The node-test part of a location step restricts the set of nodes to having a specific name or being of a specific type. The set of nodes returned by the node-test can be further filtered by applying one or more predicates which supports the usual boolean, mathematical, and string operators. For each node in the node set to be filtered, the predicate is evaluated with that node as its context node. If the predicate evaluates to true, the context node is included in the result set.

Example 3.9 Select all ECs which are of the flip-flop class and are manufactured by either Johnson Components or by the manufacturer with code M33:
/Components/Class/Component[Manufacturer/MName = 'Johnson Components' OR Manufacturer/@MCode = 'M33'][../ClassName='Flip-flop'] □

To use XPath as a foundation for the federated data model and query language we formalize the notion of an XPath expression.

Definition 3.8 (XPath Expression) An XPath expression is a function $\text{XP} : S \mapsto \mathcal{P}(S)$ where S is a set of nodes. The set of all valid XPath expressions over an XML document x is called XP_x, while the subset of XP_x that are absolute XPath expressions is called $AbsXP_x$. That is, $AbsXP_x = \{xp \in XP_x | Dom(xp) = \text{Root}(x)\}$. $RelXP_x$ is the set of expressions in XP_x that are not in $AbsXP_x$. □

4 Federating OLAP and XML

In this section we describe how links between an OLAP component and external XML components can be used to logically federate OLAP databases and XML documents as defined in Section 3.

A *link* is essentially a relationship between a dimension value in a cube and a node in an XML document. By creating a link, the user or DBA defines a sort of *cube view* containing an additional dimension. However, the actual contents of the new dimension are not defined until a query is posed. The task of creating links can be performed by executing a special "CREATE LINK" statement or it can be performed using an XML browser, e.g integrated in a visual querying tool. As will be seen, the link concept makes it easier for users to refer to XML data in OLAP queries and the mechanism also provides location transparency, since links can be changed without affecting existing queries. This is important as some types of XML documents may change their location from time to time. The fundamental linking mechanism is simply a relation between dimension values in a level and nodes in an XML document.

Definition 4.1 (Link) A link is a relation $link_L \subseteq \{(e, s) | e \in L \wedge s \in S\}$, where L is a level and S is a set of nodes. □

The basic way of specifying a link is by *enumerated linking*, which explicitly defines the relation by providing a set of three-tuples consisting of a dimension value, the XML document in which a node is to be found, and an XPath expression identifying one or more nodes in the document. Thus, one such tuple can define a number of link tuples for a single dimension value.

Definition 4.2 (Enumerated link) An enumerated link is a function $\text{EnLink} : \mathcal{P}(L \times X \times AbsXP_x) \mapsto Links$ where L is a level, X is a set of XML documents, $AbsXP_x$ is a set of absolute XPath expressions over a document $x \in X$, and $Links$ is a set of links. The resulting link relation is given by:
$\text{EnLink}(\{(e_1, x_1, locator_{x_1}), \ldots, (e_k, x_k, locator_{x_k})\}) = \{(e_i, s) | e_i \in \{e_1, \ldots, e_k\}$
$\wedge s \in locator_{x_i}(\text{Root}(x_i))\}$, where e_i is a dimension value, x_i is an XML document, $locator_{x_i}$ is an absolute XPath expression over x_i called the *locator path*. □

Example 4.1 We want to refer to the suppliers' names in the Components document when querying the Purchases database. Since the codes used for suppliers in the document are different from the ones used in the database, we have no way of identifying the links automatically. Hence, we may use an enumerated link:
$\{($"S1", "www.comp-org.org/components.xml", "/Components/Supplier[@SCode='SU13']"$)$,
$($"S3", "www.comp-org.org/components.xml", "/Components/Supplier[@SCode='SU15']"$)\}$

Note that S2 is not present in the XML document. In this case each of the tuples identify only one node in the document and the resulting link is: $Sup\_Link = \{(S1, s_1), (S3, s_3)\}$, where s_1 is the single element pointed to by: "/Components/Supplier[@SCode='SU13']" in the document "www.comp-org.org/components.xml", and similarly for s_3. □

Often, names of dimension values, or a simple transformation of the names, can be found somewhere in the nodes they should be linked to. For example, when decorating countries with their populations, it is likely that the country names can be used to identify the populations. However, there may may not be an exact match between the name of a dimension value and a node in the XML document. For example, dimension values may be full country names, while only abbreviated country codes, such as UK or US, are found in the XML document.

Enumerated linking is only necessary in the rather special case when names of dimension values cannot easily be mapped to nodes in the linked XML document, or the nodes occur in different documents. The former situation may e.g. be necessary if also historical population figures are present in the document and the link should only point to the most resent figure. More often *natural links* can be used as a shorthand. Here, the idea is to specify a level and a set of nodes in an XML document, and use the dimension values to identify one or more of these nodes. Optionally, an *alias function* may be supplied, mapping each dimension value to an alias which is used to identify the XML nodes. The set of nodes is defined for each level by a URI identifying the XML document and two XPath expressions. The first one identifies the nodes to which the link will point, and the second one is used to select the subset of these nodes that are linked to the given dimension value. The reason for using two XPath expressions is to facilitate the common case that a link must point to a subtree, but the subtree is identified by some lower node in the subtree. It is not necessary to use two expressions since XPath expressions allow you to move up the tree as well as down, but it makes it easier to use the links.

Definition 4.3 (Natural link) Assume a domain $Aliases$ of string values for XML nodes and an injective function $Alias : L \rightarrow Aliases$, mapping dimension values from L to strings in $Aliases$.

A natural link is a function: $\text{NatLink} : LS \times X \times AbsXP_x \times RelXP_x \times Alias \mapsto Links$. The resulting link relation is given by $\text{NatLink}(L, x, base, locator, alias) = \{(e, s) | e \in L \wedge s \in base(\text{Root}(x)) \wedge \exists s' \in locator(s)(\text{StrVal}(s') = alias(e))\}$, where L is a level, x is an XML document, $base \in AbsXP_x$ identifies the nodes, and $locator \in RelXP_x$ identifies the nodes being compared to dimension values in L. □

If no *alias* function is necessary, it may be omitted, i.e. the *identity* function is assumed.

Example 4.2 If we want to create a link between the ECs in the Purchases database and those in the Components document we can make a natural link, since the same codes are used in both places.

From the natural link: ("EC", "www.comp-org.org/components.xml", "/Components/Supplier/Class/Component", "@CompCode", i_{EC}), where $i_{EC}(e) = e$ is the identity function, we create the link $EC\_Link = \{(EC1234, s_1), (EC1234, s_2), (EC1235, s_3)\}$. s_1 is the first element in the Components document with CompCode="EC1234", s_2 is the second element with CompCode="EC1234", and s_3 is the single element with CompCode="EC1235". □

A flexible linking mechanism must allow both dimension values and nodes to occur more than once in the same link. The *cardinality* of a link $link$ between a level L and an XML document x can be either [1-1], [n-1], [1-n], or [n-n]. A link is [1-1] if $\|link\| = \|\pi_L(link)\| = \|\pi_x(link)\|$, where π denotes relational projection and $\|R\|$ denotes the cardinality of relation R. Similarly, the cardinality of $link$ is [n-1] if $\|link\| =$

$\|\pi_L(link)\| > \|\pi_x(link)\|$, [1-n] if $\|link\| = \|\pi_x(link)\| > \|\pi_L(link)\|$, and [n-n] if $\|link\| > \|\pi_x(link)\|$ and $\|link\| > \|\pi_L(link)\|$. We use the abbreviations [-1] to denote [1-1] or [n-1] and [-n] to denote [1-n] or [n-n]. Note that these cardinalities are not specified in any way, but are merely properties of the links.

Example 4.3 Sup_Link is [1-1] and EC_Link is [1-n]. □

To allow references to XML data in OLAP queries, links are used to define *level expressions*. A level expression consists of a starting level L, a link $link$ from L to nodes in one or more XML documents, and a relative XPath expression xp which is applied to these nodes to identify new nodes.

Definition 4.4 (Level expression) A level expression $L/link/xp$, where L is a level, xp is an XPath expression, and $link$ is a link from L, defines a link $E = \{(e, s) | e \in L \wedge \exists s'((e, s') \in link \wedge s \in xp(s'))\}$. The cardinality of a level expression is the link cardinality of E. Also, we say that a level expression *covers* its starting level if $L = \pi_L(E)$. If the starting level is not covered some facts may not be linked to any nodes. We will refer to such tuples as *unconnected* facts.

To simplify link usage we assume a function DefaultLink : $L \mapsto Links$, where L is a set of levels and $Links$ is a set of links. The function returns the default link for a given level. □

Example 4.4 The level expression “EC/EC_Link/CompCode” is [1-n] and does not cover its starting level, since “EC2345” is not mentioned in the Components document. □

Assuming that DefaultLink(EC) returns “EC_Link” the above level expression can be written “EC/CompCode”. In the following we assume that EC_Link and Sup_Link are default links for the EC and $Supplier$ levels, respectively.

With the linking mechanism in place, we can now define the federated data model consisting of a cube, a set of XML documents, and a set of links between them. We only consider one cube since multiple cubes can be handled by creating a view over the cubes.

Definition 4.5 (Federation) A federation $\mathcal{F}$ of a cube C and a set of XML documents X is a three-tuple: $\mathcal{F} = (C, Links, X)$ where $Links$ is a set of links between levels in C and documents in X. □

When it is clear from the context, we will refer to $\mathcal{F}$ as a cube, meaning the cube part of the federation $\mathcal{F}$.

Example 4.5 The cube in Example 3.3 for the Purchases database, the Components document, and the links Sup_Link and EC_Link is a federation. We will refer to this as the Purchases federation in the following. □

5 Querying Federations

In this section we present an algebra over federations and use this to define the semantics of the new federation query language SQL$_{XM}$. Hence, the focus is on the formal foundation for the querying of federations, while the more practical aspects of this are described in Section 7 and 8. SQL$_{XM}$ can be used to decorate a cube with XML data, group by XML data, and use XML data for selection. The semantics are given by extending the cube algebra to federations, providing a decoration operator, a generalized projection operator, and a selection operator. The algebra is closed, as all operators work on a federation and also return a federation.

5.1 Decoration

It is often useful to provide supplementary information for one or more levels in the result of an OLAP query. This is commonly referred to as *decorating* the result [GBLP96]. For example, products could be decorated with a competitor's prices for the same products, employees with their addresses, or suppliers with their contact person. Such information will often be available to the relevant people as Web pages on the Internet, an intranet, or an extranet. Also, this kind of information will most likely not be stored in an OLAP database because it either changes too frequently, was not expected to be used, is owned by someone else, or for some other reason. The solution suggested in this paper is to allow OLAP queries to reference external XML data using level expressions in the SELECT clause. In Section 5.2 we consider how to use level expressions in the GROUP BY clause.

Example 5.1 Let "AllTimePurchases" be the Purchases cube aggregated to the EC and Supplier levels. The fact table of this cube is shown in Table 5(a). Given the federation consisting of the "AllTimePurchases" cube, the Components document, and the links defined above, the following query decorates all ECs with their descriptions from the Components document:

```
SELECT  Cost, Supplier, EC, EC/Description
FROM    AllTimePurchases
```

□

There are two important problems with the use of level expressions for decoration which are related to the problems with non-strict and non-covering hierarchies as discussed earlier. First, a dimension value may be associated with more than one node, i.e. when the level expression has cardinality [-n]. Second, some dimension values may not be associated with any nodes at all, which is the case if the level expression does not cover its starting level.

The first problem allows for a number of different *decoration semantics*. Consider the following example:

Example 5.2 From the query in Example 5.1 we could get the result shown in Table 5(b), where a fact is created for each different description node resulting from the level expression. Another possibility is the result shown in Table 5(c), where an arbitrary node is picked and at most one fact is created for each EC. A third possibility is shown in Table 5(d), where all description nodes are concatenated. In all cases we use "N/A" to indicate that no description is found for an EC. □

Cost	Supplier	EC
2940	S1	EC1234
6900	S3	EC1234
32050	S2	EC1235
9480	S3	EC2345

(a)

Cost	Supplier	EC	Description
2940	S1	EC1234	D-type flip-flop
2940	S1	EC1234	16-bit flip-flop
6900	S3	EC1234	D-type flip-flop
6900	S3	EC1234	16-bit flip-flop
32050	S2	EC1235	16-bit flip-flop
9480	S3	EC2345	N/A

(b)

Cost	Supplier	EC	Description
2940	S1	EC1234	D-type flip-flop
6900	S3	EC1234	D-type flip-flop
32050	S2	EC1235	16-bit flip-flop
9480	S3	EC2345	N/A

(c)

Cost	Supplier	EC	Description
2940	S1	EC1234	D-type flip-flop, 16-bit flip-flop
6900	S3	EC1234	D-type flip-flop, 16-bit flip-flop
32050	S2	EC1235	16-bit flip-flop
9480	S3	EC2345	N/A

(d)

Figure 5: The fact table for the AllTimesPurchases cube and different decorations of it.

The problem is which of the nodes to use for decoration when a level expression returns more than one node. Several solutions are possible including picking one arbitrarily, using the first one, concatenating all different nodes, or using all the nodes thereby creating duplicated facts. Note that concatenating the nodes from an XML document is always possible since all nodes have a string value, though the concatenated string may not make sense to a user. Duplicating facts means that further aggregation may give an incorrect result. This is the case when grouping over decoration values, whereas grouping over other values produces a correct result. Note that the *any/all selection semantics* described in Section 3.2 are different from these *decoration semantics*. (However, as will be seen in Section 5.3 the two concepts are related.)

The second problem with the use of level expressions for decoration is how to handle expressions that do not cover its starting level. The solution used in Example 5.2 is to add a special N/A value, indicating that no nodes are available. Alternatives are to remove the facts that are not linked to any nodes or to require the level expression to cover its starting level. Removing the unconnected facts would lead to a non-summarizable result, whereas requiring all values in the starting level to be covered would reduce the practical usefulness of decoration significantly. Thus, we propose to add a special value for all unconnected facts.

Since different semantics are needed in different situations, we allow the user to choose between different types of semantics when decorating a cube with XML data. We have chosen the following because we believe they can all be useful under different circumstances:

ANY: Use an arbitrary node. This is useful when summarizability should be preserved and no node is more important than another, as might be the case e.g. when decorating suppliers with a contact person.

CONCAT: Use the concatenation of string values for all different nodes. Useful when summarizability should be preserved and all nodes are needed, e.g. when decorating products with text descriptions.

ALL: Use all different nodes, possibly duplicating facts. Useful when the decoration is used for grouping or selection.

Although we only consider these three semantics, others could be useful in some situations. For example, all nodes could be used in the CONCAT and ALL semantics, including the ones that are identical. The purpose of this would most likely be to find the number of identical ones, e.g. the number of times an EC occurs in a list of sales. This value could then be used to calculate new measure values in the cube, e.g. the total sales per EC by multiplying the number of times an EC was sold with the price of a single EC. Also, semantics preserving the document order of XML documents could sometimes be useful. However, the preservation of document order is outside the scope of this paper.

The user specifies the semantics of a decoration by giving a semantic modifier in the level expression as in EC[ANY]/EC_Link/Description. If no semantic modifier is specified, ANY semantics are assumed as the default. Notice that if the cardinality of the level expression is [-1], then decoration with the three semantics will produce the same result.

Example 5.3 The query in Example 5.1 actually results in Table 5(c) since ANY is the default. Table 5(b) and Table 5(d) are the results of these two queries, respectively:

```
SELECT   SUM(Cost), Supplier, EC, EC[ALL]/Description
FROM     AllTimePurchases

SELECT   SUM(Cost), Supplier, EC, EC[CONCAT]/Description
FROM     AllTimePurchases
```

□

We decorate a cube by adding a new dimension containing only the top level and a level containing all the decoration values. Different approaches could be to attach the decoration data as special attributes of the decorated values [Leh98, CT98, TSC99], create a new level in the same dimension as the starting level, or to keep the decorated data in an external component [PSGJ00]. Our approach has the advantage that the external data can easily be used for aggregation and selection because the decoration data is incorporated into the cube as any other dimensional data. Furthermore, aggregation is still possible in the original dimensions, as these are not changed by decoration.

With ALL decoration semantics this approach would create new facts in the fact table. Since this is an awkward behavior for a decoration operator, decoration with ALL semantics adds a new dimension with the same bottom values as the dimension containing the decorated level. The decoration data becomes dimension values in the new dimension with these bottom values as children. Thus, each bottom value rolls up to the value that decorates it. This roll-up and the possible generation of new facts is then handled by the generalized projection operator which is defined in Section 5.4.

Example 5.4 The result of decorating Table 5(a) with ALL semantics without rolling up to the decoration level is shown in Figure 6. To avoid creating new facts in the fact table, the EC values are simply duplicated, while the decoration values are added to the new dimension above these EC' values. Thus, the new facts are not created until the cube is rolled up to the Description level. Note that the new dimension is non-strict since "EC1234" has two different descriptions. □

Cost	Supplier	EC	EC'
2940	S1	EC1234	EC1234
6900	S3	EC1234	EC1234
32050	S2	EC1235	EC1235
9480	S3	EC2345	EC2345

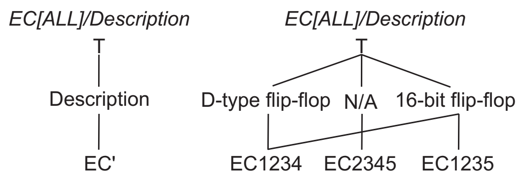


Figure 6: The fact table, new dimension schema and new dimension resulting from decoration with ALL semantics.

The three semantics are formally defined in the following. The ANY semantics is defined as:

Definition 5.1 (Decoration with ANY semantics) Decoration with ANY semantics δ_{ANY} of a federation $\mathcal{F} = (C, Links, X)$ is defined as: $\delta_{\text{ANY}[L_z, link, xp]}(\mathcal{F}) = (C', Links', X')$ where $L_z \in D_z$, $link \in Links$ is a link from L_z to X and xp is an XPath expression over X. The new federation is given by: $Links' = Links$, $X' = X$, $C' = (N', D', F')$, and $N' = N$. A new dimension is added if it is not already present: $D' = \{D_1, \ldots, D_n\} \cup \{D_{n+1}\}$ where $D_{n+1} = (L_{D_{n+1}}, E_{D_{n+1}})$. Here, $L_{D_{n+1}} = (LS_{n+1}, \sqsubseteq_{n+1}, \top_{n+1}, L_{xp})$, where $LS_{n+1} = \{\top_{n+1}, L_{xp}\}$, and $\sqsubseteq_{n+1} = \{(L_{xp}, \top_{n+1})\}$.

Let $U = \{e \in L_z | \forall (e, s) \in link(xp(s) = \emptyset) \vee \nexists s((e, s) \in link)\}$ be the set of dimension values in L_z that either has no corresponding value in $xp(s)$ or is not linked to anything via $link$. Also, let $L = \{e_{xp} | \exists (e, s) \in link(e_{xp} = \text{StrVal}(s_i)$ for some $s_i \in xp(s)\}$. Then the new level containing the decoration values is given by:

$$L_{xp} = \begin{cases} L \bigcup \{\text{N/A}\} & \text{for } U \neq \emptyset \\ L & \text{otherwise.} \end{cases}$$

Furthermore, $E_{D_{n+1}} = (L_{xp} \bigcup \top_{n+1}, \sqsubseteq_{D_{n+1}})$, where $\sqsubseteq_{D_{n+1}} = \{(e_{xp}, \top_{D_{n+1}}) | e_{xp} \in L_{xp} \wedge \top_{D_{n+1}} \in \top_{n+1}\}$.

If $D_{n+1} \in D$ then $F' = F$. Otherwise, the new fact table is given by $F' = \{t_h\}$, where for all $e_{xp} \in L_{xp}$ and $(e_{\perp_1}, \ldots, e_{\perp_n}, v_1, \ldots, v_m) \in F$:

$$t_h = \begin{cases} (e_{\perp_1}, \ldots, e_{\perp_n}, e_{xp}, v_1, \ldots, v_m) & \text{if } \exists e_{\perp_z} \in \{e_{\perp_1}, \ldots, e_{\perp_n}\}(\exists (e, s) \in link \\ & \quad (e_{\perp_z} \sqsubseteq_{D_z} e \wedge \exists s' \in xp(s)(\text{StrVal}(s') = e_{xp})) \\ & \quad \vee \exists e \in U(e_{\perp_z} \sqsubseteq_{D_z} e \wedge e_{xp} = \text{N/A})) \\ (e_{\perp_1}, \ldots, e_{\perp_n}, e_{xp}, \text{NULL}, \ldots, \text{NULL}) & \text{otherwise.} \end{cases}$$

For all measures M_h in C' the aggregation type is: $\text{AggType}(M_h, D_{n+1}) = \text{AggType}(M_h, D_z)$. □

Intuitively, only two things are changed when decorating a cube: A new dimension is added and the fact table is updated to reflect this. However, since the cube definition does not allow duplicate dimensions, no changes are made if an identical dimension already exists in the cube. The new dimension contains only the decoration level and the top level. The new dimension values in the decoration level are created from an arbitrarily chosen node found by following the link from the starting level and then applying the XPath expression. If one or more values in the starting level does not produce any decoration values the special N/A value is used instead. The new fact table is created from the cartesian product of the dimension values from the old fact table and the new decoration values. Measure values are replaced with NULL values such that no facts are duplicated.

Example 5.5 The federation $AllTimePurchases_{Fed} = (AllTimePurchases_{Cube}, \{EC\_Link, Sup\_Link\}, \{\text{“www.comp-org.org/components.xml”}\})$ is decorated with EC descriptions as follows:

$\delta_{\text{ANY}[EC, EC\_Link, \text{“Description”}]}(AllTimePurchases_{Fed}) = AllTimePurchases_{Fed}{}'$.

$AllTimePurchases_{Fed}{}'$ contains the same links and XML document while the $AllTimePurchases_{Cube}$ cube is extended with a new dimension $D_{Description}$. Letting Des denote $Description$ we have that $D_{Des} = (L_{D_{Des}}, E_{D_{Des}})$. Here, $L_{D_{Des}}$ is the poset $(\{\top_{Des}, L_{Des}\}, \sqsubseteq_{Des}, \top_{Des}, L_{Des})$, where $\sqsubseteq_{Des} = \{(L_{Des}, \top_{Des})\}$. Also, $E_{D_{Des}} = (\{\top, \text{“D-type flip-flop”}, \text{“16-bit flip-flop”}, \text{“N/A”}\}, \sqsubseteq_{D_{Des}})$, where $\top$ is the parent of the other values in the ordering. The new fact table is shown in Table 5(c). □

The only difference between ANY and CONCAT semantics is in the definition of the decoration values. For ANY semantics we pick one of the values, for CONCAT semantics we concatenate all the different values.

Definition 5.2 (Decoration with CONCAT semantics) δ_{CONCAT} is defined as for ANY semantics except for the following change in the definition of L:

For each $e \in L_z$ let $S_e = \{s | \exists s'((e, s') \in link \wedge s \in xp(s'))\}$. Then $L = \{e_{xp} | \exists e \in L_z(e_{xp} = \text{Concat}(\text{StrVal}(s_1), \ldots, \text{StrVal}(s_k))$ for $s_i \in S_e)\}$, where Concat is a function concatenating a set of strings. □

For ALL semantics a new dimension is added containing the same bottom values as the dimension to which the starting level belongs. The decoration values are then inserted between the top value and the bottom values in the new dimension. A decoration value becomes the parent of a bottom value if the bottom value is the child of a value in the starting level that is linked to the decoration value.

Definition 5.3 (Decoration with ALL semantics) δ_{ALL} is defined as for ANY semantics except for the following changes:

- The new poset of levels consists of three levels: $L_{D_{n+1}} = (LS_{n+1}, \sqsubseteq_{n+1}, \top_{n+1}, \bot_{n+1})$ where $LS_{n+1} = \{\top_{n+1}, \bot_{n+1}, L_{xp}\}$ and $\sqsubseteq_{n+1} = \{(\bot_{n+1}, L_{xp}), (L_{xp}, \top_{n+1}), (\bot_{n+1}, \top_{n+1})\}$.
- The new dimension values of L_{xp} is $L = \{e_{xp} | \exists (e, s) \in link(\exists s' \in xp(s)(e_{xp} = \text{StrVal}(s')))\}$. The dimension values of the new bottom level are given by: $\bot_{n+1} = \bot_z$.
- The new ordering of dimension values is $E_{D_{n+1}} = (\bot_{n+1} \cup L_{xp} \cup \top_{n+1}, \sqsubseteq_{D_{n+1}})$, where $\sqsubseteq_{D_{n+1}} = \{(e_{\bot_{n+1}}, \top_{D_{n+1}}) | e_{\bot_{n+1}} \in \bot_{n+1} \wedge \top_{D_{n+1}} \in \top_{n+1}\} \bigcup \{(e_{xp}, \top_{D_{n+1}}) | e_{xp} \in L_{xp} \wedge \top_{D_{n+1}} \in \top_{n+1}\} \bigcup \{(e_{\bot_{n+1}}, e_{xp}) | e_{\bot_{n+1}} \in \bot_{n+1} \wedge e_{xp} \in L_{xp} \wedge (\exists e, e_{\bot_z}((e, s) \in link \wedge e_{\bot_{n+1}} = e_{\bot_z} \wedge e_{\bot_z} \sqsubseteq_{D_z} e \wedge \exists s_j \in xp(s)(e_{xp} = \text{StrVal}(s_j))) \vee \exists e, e_{\bot_z}(e \in U \wedge e_{\bot_{n+1}} = e_{\bot_z} \wedge e_{\bot_z} \sqsubseteq_{D_z} e \wedge e_{xp} = \text{N/A}))\}$.
- The new fact table is given by $F' = \{t_h\}$ where for all $e_{\bot_{n+1}} \in \bot_{n+1}$ and $(e_{\bot_1}, \ldots, e_{\bot_n}, v_1, \ldots, v_m) \in F$:

$$t_h = \begin{cases} (e_{\bot_1}, \ldots, e_{\bot_n}, e_{\bot_{n+1}}, v_1, \ldots, v_m) & \text{if } \exists e_{\bot_z} \in \bot_z(e_{\bot_z} \in \{e_{\bot_1}, \ldots, e_{\bot_n}\} \wedge e_{\bot_z} = e_{\bot_{n+1}}) \\ (e_{\bot_1}, \ldots, e_{\bot_n}, e_{\bot_{n+1}}, \text{NULL}, \ldots, \text{NULL}) & \text{otherwise.} \end{cases}$$

□

Intuitively, the new bottom level contains the same dimension values as the bottom level of the dimension to which the starting level belongs. The new decoration level contains a dimension value for each distinct node found by following the link and applying the user defined XPath expression. The fact table is created from the dimension values from the old fact table and the new bottom values of the decoration dimension. No measure values are duplicated since they only exist for tuples containing the same bottom value twice. All other measure values are set to NULL . Thus, the ALL decoration operator only associates values in the starting level with the right decoration values through their common bottom values. To actually present the decoration along with the other levels, the new decoration dimension must be aggregated to the level containing the new decoration values. In the following we discuss how to perform aggregation over federations.

5.2 Extending Grouping to Federations

Allowing level expressions in the GROUP BY clause makes it possible to group by data from XML documents, without having to physically store this data in the OLAP database. For example, product prices will often be available from a supplier's Web page or an e-marketplace. These up-to-date prices can then be used to group products in an OLAP product database without having to store the prices.

Example 5.6 The following query groups ECs after their text descriptions from the Components document.

```
SELECT      SUM(Cost), EC[ALL]/Description
FROM        Purchases
GROUP BY    EC[ALL]/Description
```
□

GROUP BY queries with level expressions are evaluated in two steps. First, the cube is decorated as described in the previous section. Second, aggregation is performed by using the already defined generalized projection Π_{Cube} on the new cube.

Example 5.7 The above query is evaluated by first decorating the Purchases cube resulting in the fact table shown in Table 5(b), and then grouping by "Description" using Π_{Cube}. □

When decorating the cube, the new decoration dimension may be non-strict if ALL semantics are used and a bottom value is decorated by more than one decoration value. This is the reason for allowing non-strictness in a cube and for handling it in the generalized projection operator as defined in Definition 3.6. Consequently, if non-strictness occurs because of the decoration and if aggregation results in duplicated facts, this is handled by setting the aggregation type to c, preventing further aggregation in that dimension. To aggregate further, the original cube must be used.

Formally, the generalized projection operator over federations is defined as follows:

Definition 5.4 (Generalized projection over federations) Let $\mathcal{F} = (C, Links, X)$ be a federation and $M_{j_1}, \ldots, M_{j_m}$ be measures in C. Also let $L_1, \ldots, L_k$ be levels in C such that at most one level from each dimension occurs. The generalized projection operator Π_{Fed} over federation $\mathcal{F}$ is then defined as:

$\Pi_{Fed[L_1,\ldots,L_k]<f_{j_1}(M_{j_1}),\ldots,f_{j_m}(M_{j_m})>}(\mathcal{F}) = (C', Links', X')$

where the new cube is $C' = \Pi_{Cube[L_1,\ldots,L_k]<f_{j_1}(M_{j_1}),\ldots,f_{j_m}(M_{j_m})>}(C)$.

Links for which the starting level no longer exists are removed from the resulting federation. That is: $Links' = \{link \in Links | \exists L \in C'(\exists(e,s) \in link(e \in L))\}$. XML documents to which no links refer are also removed: $X' = \{x \in X | \exists link \in Links'(\exists(e,s) \in link(s \in \text{Nodes}(x)))\}$. □

5.3 Extending Selection to Federations

XML data can also be used to perform selection over cubes. This makes it possible e.g. to view only products where a certain supplier is cheaper than another supplier by referring to their Web pages. The idea adopted here is to allow level expressions in WHERE and HAVING predicates in places where levels can already be used. For example, level expressions can be compared to constants, levels, measures, or other level expressions.

Example 5.8 Show component costs by supplier and EC but only those available for less than 3.00 euro.

```
SELECT      SUM(Cost), Supplier, EC
FROM        Purchases
WHERE       EC/UnitPrice[@Currency='euro'] < 3.00
GROUP BY    Supplier, EC
```

□

As discussed in Section 3.2 selection semantics are also affected by the cardinality and covering properties of level expressions. As for selection over cubes, we handle this by using *any* semantics.

Selection over federations is evaluated by first decorating with all the level expressions mentioned in the predicate. The resulting federation is then sliced using the selection operator, and finally, the new decoration dimensions are removed again. The selection operator simply applies the cube selection operator to the cube part of the federation since the link and XML parts should not be affected by selection. ALL semantics are used for the decorations to make sure that all decoration values are available. This is important since *any* selection semantics are used in predicates, and thus, a predicate may be satisfied by any of the decoration values. No facts are duplicated since the ALL decoration is never actually rolled up to the decoration level. The roll-up is handled entirely by the cube selection operator.

Example 5.9 Show only components that are manufactured by the supplier.

```
SELECT      SUM(Cost), Supplier, EC
FROM        Purchases
WHERE       EC/Manufacturer/MName = EC/../../Suppliers/SName
GROUP BY    Supplier, EC
```

This query is evaluated by first decorating with the level expressions EC/Manufacturer/MName and EC/../../Suppliers/SName using the ALL semantics. This results in two new columns EC$'$ and EC$''$ in the fact table both duplicating the EC level. A new predicate is then constructed rolling up to the decoration level: "Manufacturer/MName"(EC$'$) = "../../Suppliers/SName"(EC$''$), and this is used to select a part of the fact table. Finally, the two new columns are removed again. □

Formally, selection over federations is defined as follows:

Definition 5.5 (Selection over federations) Let $\mathcal{F} = (C, Links, X)$ be a federation.

The selection operator over federations is then defined as: $\sigma_{Fed[p]}(\mathcal{F}) = (C', Links', X')$, where $Links' = Links$, $X' = X$, and the new cube is $C' = \sigma_{Cube[p]}(C)$. □

Hence, selection is performed on a federation by applying cube selection to the cube part using a predicate without the level expressions. The decoration and predicate transformation are not handled by the selection operation but instead by the mapping from SQL$_{XM}$ to the federation algebra as described in the next section.

This concludes the definition of the algebra. The three operators defined all operate on federations and result in federations, thus, the federation algebra is closed.

5.4 Semantics of the SQL$_{XM}$ Query Language

We can now define the semantics of a SQL$_{XM}$ query in terms of the federation algebra. The syntax is given in Appendix A.

Definition 5.6 (Semantics of an SQL$_{XM}$ query over a federation) In the following, let:

- $\mathcal{F} = (C, Links, X)$ be a federation,

- $\perp_{g_1}, \ldots, \perp_{g_k} \subseteq \perp_1, \ldots, \perp_n$ and $L_{g_1}, \ldots, L_{g_k}$ be levels in C such that $\perp_{g_h} \sqsubseteq_i L_{g_h}$,
- $M_{j_1}, \ldots, M_{j_l} \subseteq M_1, \ldots, M_m$ be a set of measures from C,
- $f_{j_1}, \ldots, f_{j_l}$ be aggregation functions all of which are assumed to be distributive,
- $pred_{where}$ be a predicate over levels and measures in C containing level expressions $\mathcal{L}_{w_1}/link_{w_1}/xp_{w_1}, \ldots, \mathcal{L}_{w_{p_1}}/link_{w_{p_1}}/xp_{w_{p_1}}$,
- $\mathcal{L}_{s_1}[Sem_{s_1}]/link_{s_1}/xp_{s_1}, \ldots, \mathcal{L}_{s_{p_2}}[Sem_{s_{p_2}}]/link_{s_{p_2}}/xp_{s_{p_2}}$ be level expressions, where each $\mathcal{L}_i$ is a level in C, $link_i \in Links$ is a link from $\mathcal{L}_i$, and xp_i is an XPath expression,
- $pred_{having}$ be a predicate over levels and measures in the cube after grouping, containing level expressions
  $\mathcal{L}_{h_1}/link_{h_1}/xp_{h_1}, \ldots, \mathcal{L}_{h_{p_3}}/link_{h_{p_3}}/xp_{h_{p_3}}$.

Also, we use the abbreviation $\Delta_{E_1,\ldots,E_n}(\mathcal{F}) = \delta_{E_n}(\ldots(\delta_{E_1}(\mathcal{F}))$, where $E_i = Sem_i[\mathcal{L}_i, link_i, xp_i]$.
The $\textsf{SQL}_{XM}$-query

SELECT	$f_{j_1}(M_{j_1}), \ldots, f_{j_l}(M_{j_l}), L_{g_1}(\perp_{g_1}), \ldots, L_{g_k}(\perp_{g_k})$, $\mathcal{L}_{s_1}[Sem_{s_1}]/link_{s_1}/xp_{s_1}, \ldots, \mathcal{L}_{s_{p_2}}[Sem_{s_{p_2}}]/link_{s_{p_2}}/xp_{s_{p_2}}$
FROM	$\mathcal{F}$
WHERE	$pred_{where}$
GROUP BY	$L_{g_1}(\perp_{g_1}), \ldots, L_{g_k}(\perp_{g_k}), \mathcal{L}_{s_1}[Sem_{s_1}]/link_{s_1}/xp_{s_1}, \ldots, \mathcal{L}_{s_{p_2}}[Sem_{s_{p_2}}]/link_{s_{p_2}}/xp_{s_{p_2}}$
HAVING	$pred_{having}$

can then be evaluated as

$$\begin{aligned}
&\Pi_{Fed[L_{g_1},\ldots,L_{g_k},xp_{s_1},\ldots,xp_{s_{p_2}}]<f_{j_1}(M_{j_1}),\ldots,f_{j_l}(M_{j_l})>}(\\
&\quad\sigma_{Fed[pred_{having}']}(\Delta_{\text{ALL}[\mathcal{L}_{h_1},link_{h_1},xp_{h_1}],\ldots,\text{ALL}[\mathcal{L}_{h_{p_3}},link_{h_{p_3}},xp_{h_{p_3}}]}(\\
&\quad\quad\Pi_{Fed[L_{g_1},\ldots,L_{g_k},xp_{s_1},\ldots,xp_{s_{p_2}}]<f_{j_1}(M_{j_1}),\ldots,f_{j_l}(M_{j_l})>}(\Delta_{Sem_{s_1}[\mathcal{L}_{s_1},link_{s_1},xp_{s_1}],\ldots,Sem_{s_{p_2}}[\mathcal{L}_{s_{p_2}},link_{s_{p_2}},xp_{s_{p_2}}]}(\\
&\quad\quad\quad\sigma_{Fed[pred_{where}']}(\Delta_{\text{ALL}[\mathcal{L}_{w_1},link_{w_1},xp_{w_1}],\ldots,\text{ALL}[\mathcal{L}_{w_{p_1}},link_{w_{p_1}},xp_{w_{p_1}}]}(\mathcal{F})))))))
\end{aligned}$$

The new predicates $pred_{where}'$ and $pred_{having}'$ are constructed from $pred_{where}$ and $pred_{having}$, respectively, by replacing each level expression $L/link/xp$ with the roll-up function $xp(L_\perp)$, where $L_\perp$ is the bottom level of the dimension to which L belongs. $\square$

An $\textsf{SQL}_{XM}$ query over a federation is evaluated in four major steps. First, the cube is sliced as specified in the WHERE clause, possibly requiring a decoration with XML data which is then projected away after selection. Second, the resulting cube is decorated with external XML data from the level expressions occurring in the SELECT and GROUP BY clauses. This creates a number of new dimensions in the cube. Third, all dimensions, including the new ones, are rolled up to the levels specified in the GROUP BY clause. Finally, the resulting cube is sliced according to the predicate given in the HAVING clause, which may require additional decorations. Notice that the new decoration dimensions used for selection are not mentioned in the following generalized projection and are therefore removed after use. Since these new dimensions are never aggregated to the decoration level, no changes are made to the aggregation types.

Example 5.10 Calculate costs by supplier, class and EC description, decorated with the supplier names. The result should only reflect purchases of ECs which are manufactured by the manufacturer listed in the Components document and are supplied by S2 or S3. Also, we only want the groups with a total cost of more than 7.000.

SELECT	SUM(Cost), Supplier, Class(EC), Supplier/SName, EC[ALL]/Description
FROM	Purchases
WHERE	EC/Manufacturer/@MCode = Manufacturer(EC) AND Supplier IN (S2, S3)
GROUP BY	Supplier, Class(EC), Supplier/SName, EC[ALL]/EC_Link/Description
HAVING	SUM(Cost) $>$ 7.000

This query is evaluated as follows:

$$\sigma_{Fed[\text{SUM}(Cost)>7.000]}(\Pi_{Fed[Supplier,Class,SName,Description]<\text{SUM}(Cost)>}(\\ \delta_{\text{ANY}[Supplier,Sup\_Link,SName]}(\delta_{\text{ALL}[EC,EC\_Link,Description]}(\\ \sigma_{Fed[Manufacturer/@MCode(EC)=Manufacturer(EC)\text{ AND }Supplier\text{ IN }(S2,S3)]}(\\ \delta_{\text{ALL}[EC,EC\_Link,Manufacturer/@MCode]}(Purchases))))))$$

Notice that no generalized projection is needed after the last selection since it does not refer to any decorations that must be removed. □

As can be seen from this example the resulting algebraic expression can be optimized in several ways. For instance, a partial aggregation can often be performed before the first decoration, to reduce the size of intermediate results. This and other optimizations are discussed in the following sections.

6 Algebraic Transformation Rules

The optimization approaches presented in this paper cover both heuristic and cost based techniques. In Section 8 we will discuss heuristic optimization of SQL$_{XM}$ queries and in Section 9 a cost based approach is considered. To provide a basis for these techniques we present here a collection of transformation rules for the federated algebra. Most of these rules involve the decoration operation and have, to the best of our knowledge, not been considered elsewhere. The remaining rules are similar to those for Extended Relational Algebra (ERA), i.e. duplicate sensitive relational algebra with aggregations [GdB94, GHQ95]. Since most ERA rules have an equivalent rule in SQL$_{XM}$ Algebra, the list of rules is not complete, but cover rules that are important for optimizing federation queries. We consider only distributive aggregation functions, which is unproblematic since all the widely used functions are distributive or can be expressed in terms of such functions. For example, SUM, MIN, MAX are distributive, COUNT = SUM(1), and AVG can be expressed in terms of SUM and COUNT.

The full list of transformation rules are shown in Table 2. The rules are presented formally in the following by first giving an intuitive description of the rule, possibly illustrated by an example, then stating the rule formally, and finally arguing for the validity of the rule. The rules are grouped after the operators they involve.

In the formal presentation left-to-right rules are denoted by $\rightarrow$, while bidirectional rules are denoted by $\leftrightarrow$. We begin by stating a fundamental equivalence property of the decoration operation:

Theorem 6.1 If a cube $\mathcal{F}$ has already been decorated with a decoration $\delta_{S[L,link,xp]}$, all subsequent decorations with $\delta_{S[L,link,xp]}$ can be ignored. That is:

$$\delta_{S[L,link,xp]}(\mathcal{F})) \leftrightarrow \mathcal{F}$$

Proof outline: This follows from Definition 5.1, 5.2, and 5.3, which states that if an identical decoration dimension is already present, it is not added again. Hence, the decoration has no effect and can be removed or added without changing the cube. □

In the formal definitions let

No.	Description
6.1	Redundant Decoration Above Generalized Projection
6.2	Redundant Decoration Below Generalized Projection
6.3	Commutativity of Decoration and Generalized Projection
6.4	Pushing Generalized Projection Below Decoration
6.5	Commutativity of Selection and Decoration
6.6	Inlining of Decoration in Selection
6.7	Commutativity of Selection and Generalized Projection
6.8	Pushing Generalized Projection Below Selection
6.9	Commutativity of Generalized Projection and Selection with References to Measures
6.10	Commutativity of Decorations
6.11	Cascade of Selections
6.12	Commutativity of Selections
6.13	Cascade of Generalized Projections
6.14	Redundant Generalized Projection
6.15	Cascade of Decorations
6.16	Pushing Generalized Projection Below Decorations and Selections

Table 2: Transformation rules for the federation algebra.

- $E = L[S]/link/xp$ be a level expression, and $L_{xp,\perp}$ and L_{xp} the bottom and decoration level of the dimension resulting from decoration with E, respectively,
- $\mathcal{L}$ be a set of levels from different dimensions,
- $F(M) = \{f_1(M_1), \ldots, f_l(M_l)\}$ be a set of aggregation functions applied to measures,
- Dim : $Levels \rightarrow Dimensions$ be a function that returns the dimension to which a level belongs.
- Dims : $\mathcal{P}(Levels) \times Predicates \rightarrow \mathcal{P}(Dimensions)$ be a function that given a set of levels, and a predicate, returns the set of dimensions to which the levels in the set and in the predicate belong,
- Levs : $Dimension \times \mathcal{P}(Levels) \times Predicates \rightarrow \mathcal{P}(Levels)$ be a function, that given a dimension D, a set of levels $\mathcal{L}$, and a predicate θ returns the set of levels from D that are either contained in $\mathcal{L}$ or are referenced in θ,
- Max : $\mathcal{P}(Levels) \rightarrow Levels$ be a function that given a set of levels from the same dimension returns the uppermost level,
- MaxStrict : $\mathcal{P}(Levels) \rightarrow Levels$ be a function that given a set of levels from the same dimension returns the uppermost such level that do not introduce non-strictness when rolled up to from the bottom level,

For improved readability, we omit the subscript *Fed* for operators in the federation algebra, since no confusion can occur.

6.1 Rules Involving Decoration and Generalized Projection

According to Theorem 6.1, a cube can only be decorated with the same decoration once. Thus, a decoration can be removed or added above a generalized projection (GP) that already includes the decoration in the list of GROUP BY levels.

Example 6.1 $\delta_{\text{ALL}[EC,EC\_Link,Description]}(\Pi_{[EC,Supplier,Description]<\text{SUM}(Cost)>}(\delta_{\text{ALL}[EC,EC\_Link,Description]}$ $(Purchases)))$ is equivalent to $\Pi_{[EC,Supplier,Description]<\text{SUM}(Cost)>}(\delta_{\text{ALL}[EC,EC\_Link,Description]}(Purchases))$ because the cube has already been decorated with Description. □

Rule 6.1 (Redundant Decoration Above Generalized Projection) If a level L_{xp} resulting from decoration with the level expression $L[S]/link/xp$ is already present in $\mathcal{L}$ and L' occurs in $\mathcal{L}$ such that $L' \sqsubseteq L$ the following holds:

$$\delta_{S[L,link,xp]}(\Pi_{[\mathcal{L}]<F(M)>}(\mathcal{F})) \leftrightarrow \Pi_{[\mathcal{L}]<F(M)>}(\mathcal{F})$$

Reasoning: This rule follows from Theorem 6.1. □

A decoration can be removed if it occurs below a GP that does not include it in the list of GROUP BY levels.

Example 6.2 $\Pi_{[EC,Supplier]<\text{SUM}(Cost)>}(\delta_{\text{ALL}[EC,EC\_Link,Description]}(Purchases))$ is equivalent to $\Pi_{[EC,Supplier]<\text{SUM}(Cost)>}(Purchases))$ because the decoration with Description has no effect above the GP. □

Rule 6.2 (Redundant Decoration Below Generalized Projection) If $L_{xp} \notin \mathcal{L} \wedge L_{xp,\perp} \notin \mathcal{L}$ the following holds:

$$\Pi_{[\mathcal{L}]<f(M)>}(\delta_{S[L,link,xp]}(\mathcal{F})) \leftrightarrow \Pi_{[\mathcal{L}]<f(M)>}(\mathcal{F})$$

Reasoning: If the decoration does not occur in the GP it is projected away before it is used and can be removed without changing the resulting cube. □

Decoration and GP operations are commutative if the cube resulting from the GP contains the starting level of the decoration. The starting level or a level below it must be present when pushing a GP below a decoration, because otherwise it would not be possible to decorate the cube after aggregation has been performed. This requirement is always satisfied when pushing a decoration below a GP. Two different cases may occur depending on whether or not the decoration being pushed down has been applied before and is already present in the GP's GROUP BY levels. If it is already present then the previous decoration must be preserved instead of removing it and decorating again. This is necessary because the original decoration may be an ALL decoration, that has been rolled up to the decoration level and thereby prohibited further aggregation. If the decoration has not been applied before, the bottom level of the decoration dimension must be included in the GP's GROUP BY levels after pushing down the decoration. This is necessary because decoration always adds the bottom level of the decoration dimension to the fact table. Notice that if a decoration being pushed down is already present in the GP's GROUP BY levels the commutativity holds because the decoration is redundant both above and below the GP.

Example 6.3 Assume a new link Class_Link from the Class level to the Class nodes in the Components document. Then $\delta_{\text{ALL}[Class,Class\_Link,ClassName]}(\Pi_{[EC,Country]<\text{SUM}(Cost)>}(Purchases))$ is equivalent to $\Pi_{[EC,Country,EC']<\text{SUM}(Cost)>}(\delta_{\text{ALL}[Class,Class\_Link,ClassName]}(Purchases))$ where EC' is the bottom level of the new ClassName dimension. In both cases the resulting cube has bottom levels EC, Country and EC′, as well as the measure Cost. Notice that the starting level Class or some level below it (in this case EC) must occur in the GP's GROUP BY levels in order to push the GP below the decoration.

Also, $\delta_{\text{ALL}[Class,Class\_Link,ClassName]}(\Pi_{[EC,Country,EC']<\text{SUM}(Cost)>}(\delta_{\text{ALL}[Class,Class\_Link,ClassName]}$ $(Purchases))))$ is equivalent to $\Pi_{[EC,Country,EC']<\text{SUM}(Cost)>}(\delta_{\text{ALL}[Class,Class\_Link,ClassName]}$ $(\delta_{\text{ALL}[Class,Class\_Link,ClassName]}(Purchases))))$ because the topmost decoration in both expressions can be ignored. □

Rule 6.3 (Commutativity of Decoration and Generalized Projection) The following holds if a level L' occurs in $\mathcal{L}$ such that $L' \sqsubseteq L$ and $L_{xp} \notin \mathcal{L} \wedge L_{xp,\perp} \notin \mathcal{L}$:

$$\text{(a)} \quad \delta_{S[L,link,xp]}(\Pi_{[\mathcal{L}]<F(M)>}(\mathcal{F})) \leftrightarrow \Pi_{[\mathcal{L}\cup\{xp_\perp\}]<F(M)>}(\delta_{S[L,link,xp]}(\mathcal{F}))$$

Also, let the schema of $\mathcal{F}$ be $\mathcal{L}', M'$. If L' occurs in $\mathcal{L}$ such that $L' \sqsubseteq L$, $L_{xp} \in \mathcal{L} \vee L_{xp,\perp} \in \mathcal{L}$, and $L_{xp} \in \mathcal{L}' \vee L_{xp,\perp} \in \mathcal{L}'$ the following holds:

$$\text{(b)} \quad \delta_{S[L,link,xp]}(\Pi_{[\mathcal{L}]<F(M)>}(\mathcal{F})) \leftrightarrow \Pi_{[\mathcal{L}]<F(M)>}(\delta_{S[L,link,xp]}(\mathcal{F}))$$

Reasoning: Part (a) of the rule holds since neither L_{xp} nor $L_{xp,\perp}$ are in $\mathcal{L}$ and since $L_{xp,\perp}$ is added to the GP on the right-hand side. Hence, both sides have the same number of dimensions and are aggregated to the same levels. Also, aggregation types are the same, since no roll up is performed in the dimension containing the decoration data. The decoration on the left-hand side can always be performed because the level L' is in $\mathcal{L}$. Part (b) of the rule holds because according to Theorem 6.1 both decorations are redundant and can be removed. $\square$

Rule 6.3 can be generalized by introducing a new GP on the right-hand side. Intuitively, what this means is that instead of decorating a cube and then aggregating it, the cube is first aggregated to yield an intermediate cube. This cube is decorated and then further aggregated to produce the final result. The new GP aggregates the cube as much as possible, while still allowing the decoration operator to be applied. If the decoration is already present in the original cube, then it occurs in the GROUP BY levels of the new GP.

Example 6.4 $\Pi_{[EC,Country,ClassName]<\text{SUM}(Cost)>}(\delta_{\text{ALL}[Class,Class\_Link,ClassName]}(Purchases))$ is equivalent to $\Pi_{[EC,Country,ClassName]<\text{SUM}(Cost)>}(\delta_{\text{ALL}[Class,Class\_Link,ClassName]}($
$\Pi_{[EC,Country]<\text{SUM}(Cost)>}(Purchases)))$. $\square$

The intermediate cube produced by the new GP must satisfy certain requirements. First, if a level is mentioned in the original GP, this level must also be present in the intermediate cube. Second, the starting level of the decoration must be present in the intermediate cube to allow the decoration to be applied to it. Third, aggregation types in the intermediate cube should not be changed compared to the original cube because further aggregation must be allowed. These three requirements can be satisfied by choosing the proper level to roll up to in the dimension D to which the starting level of the decoration belongs. The remaining dimensions can roll up to the same levels as the original GP. If there exists a level L referenced in the original GP, which belongs to D, the uppermost level in D is chosen, such that it is possible to roll up to both L and the starting level of the decoration. If the chosen level is not L, that is, if the new GP does not roll up to the same level as the original GP, the aggregation type is not allowed to change, since further aggregation will be done by the original GP. The bottom level of D will always satisfy these requirements, and hence it is always possible to find such a level. If no level referenced in the original GP belongs to D, the starting level of the decoration can be used, provided that no aggregation types are changed. The measures in the new GP are the same as in the original GP.

Example 6.5 Instead of the Purchases cube, consider a new cube Purchases2 where the usual EC dimension is replaced with the dimension shown in Figure 7. In this dimension the relationship from EC to Type is assumed to be non-strict.

Then $\Pi_{[Month,NoOfPins,ClassName]<\text{SUM}(Cost)>}(\delta_{\text{ALL}[Class,Class\_Link,ClassName]}(Purchases2))$ is equivalent to $\Pi_{[Month,NoOfPins,ClassName]<\text{SUM}(Cost)>}(\delta_{\text{ALL}[Class,Class\_Link,ClassName]}($
$\Pi_{[Month,EC]<\text{SUM}(Cost)>}(Purchases2)))$. Note that the new GP cannot roll up to Type because further aggregation along that dimension would no longer be allowed. $\square$

Rule 6.4 (Pushing Generalized Projection Below Decoration) The following holds :

$$\Pi_{[\mathcal{L}]<F(M)>}(\delta_{S[L_s,link,xp]}(\mathcal{F})) \leftrightarrow \Pi_{[\mathcal{L}]<F(M)>}(\delta_{S[L_s,link,xp]}(\Pi_{[\mathcal{L}']<F(M)>}(\mathcal{F})))$$

where $\mathcal{L}' = \{L \in \mathcal{L} | L \notin \{L_{xp}, L_{xp,\perp}\} \wedge L \notin \text{Dim}(L_s)\} \cup \{L_{Max}\}$.
Here L_{Max} is given by:

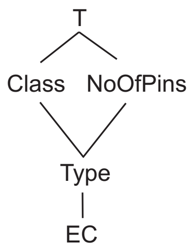


Figure 7: The new EC dimension with non-strictness between EC and Type.

$$L_{Max} = \begin{cases} \text{Max}(\mathcal{L}_{Max}) & \text{if Max}(\mathcal{L}_{Max}) \in \mathcal{L} \\ \text{MaxStrict}(\mathcal{L}_{Max}) & \text{otherwise} \end{cases}$$
where $\mathcal{L}_{Max} = \{L \in \text{Dim}(L_s) | L \sqsubseteq L_s \wedge \forall L' \in \text{Dim}(L_s)(L' \in \mathcal{L} \Rightarrow L \sqsubseteq L')\}$

If L_{xp} is already present in $\mathcal{F}$ then it is preserved in $\mathcal{L}'$.

Reasoning: The extra GP introduced on the right side, aggregates the cube to the same levels as the original GP, except for the dimension to which the starting level belongs and possibly the decoration dimension. The set of levels $\mathcal{L}'$, that the extra GP aggregates to, is constructed in such a way, that it is always possible to apply the decoration operator afterwards. Furthermore, $\mathcal{L}'$ is always possible to construct, since the bottom level of the dimension $\text{Dim}(L_s)$ is always present in $\mathcal{L}_{Max}$. Note that although the rule is explained in the left to right direction it also holds in the opposite direction because of the way $\mathcal{L}'$ is constructed.

If the decoration has already been applied in $\mathcal{F}$, it must be preserved in $\mathcal{L}'$ because the decoration may be an ALL decoration that is rolled up to the decoration level. If the dimension is non-strict this means that it cannot simply be aggregated away and added again by another decoration. Notice that preserving a decoration in the new GP is correct even when the decoration dimension is non-strict and the decoration is aggregated away by the top GP. Consider first the situation where the decoration occuring in $\mathcal{F}$ is at the bottom level. In that case the bottom level also occurs in the extra GP and hence, it performs no aggregation. The original GP aggregates to the top level which does not prohibit further aggregation because it skips the non-strictness between the bottom level and the decoration level. Thus, this situation is handled correctly. Now consider the situation where the decoration occuring in $\mathcal{F}$ is at the decoration level. Then the decoration level occurs in the extra GP and hence, it performs no aggregation in the decoration dimension. The aggregation in the original GP does indeed prohibit further aggregation, but this was also the case before introducing a new GP. Thus, this situation is also handled correctly. □

6.2 Rules Involving Decoration and Selection

Selection commutes with decoration if the selection does not refer to the decoration or if the cube has already been decorated with the same decoration.

Rule 6.5 (Commutativity of Selection and Decoration) The following holds if θ does not refer to L_{xp} or $L_{xp,\perp}$, or if $L_{xp} \in \mathcal{L}' \vee L_{xp,\perp} \in \mathcal{L}'$:

$$\sigma_\theta(\delta_{S[L,link,xp]}(\mathcal{F})) \leftrightarrow \delta_{S[L,link,xp]}(\sigma_\theta(\mathcal{F}))$$

where the schema for $\mathcal{F}$ is $\mathcal{L}', M'$.

Reasoning: Decoration only affects one dimension and thus, the truth value of the predicate is only affected if that dimension is mentioned in the predicate. Since decoration never changes the number of facts, references to measures are allowed in the predicate. Notice that the condition is always satisfied in the right to left direction. Also, observe that selections are constructed such that they never refer to the bottom level of a decoration (see Section 5.4). □

A decoration can be integrated into a predicate by creating a more complex predicate. This technique is called *inlining*. If a predicate contains references to a level expression, a new predicate can be constructed instead that contains only references to constants. In the general case the predicate can be very large, but for many simple predicates and if the number of values is small, this is indeed a practical solution. This technique is generally applicable and is explained in more detail in Section 9.2 and Appendix B, where the size of the resulting predicate is also discussed.

Example 6.6 The predicate *EC/Description = '16-bit flip-flop'* can be transformed to *EC IN (EC1234, EC1235)* because EC1234 and EC1235 has Description nodes equal to "16-bit flip-flop". □

After inlining the decoration can be moved up above the selection using Rule 6.5 because it no longer refers to the decoration.

Rule 6.6 (Inlining of Decoration in Selection) If the predicate θ contains references to the level expression $E = L[S]/link/xp$, the following holds:

$$\sigma_\theta(\delta_{S[L,link,xp]}(\mathcal{F})) \leftrightarrow \sigma_{\theta_{xp}}(\delta_{S[L,link,xp]}(\mathcal{F}))$$

where θ_{xp} no longer refers to E.
Reasoning: This is possible because the modified predicate is expressed in terms of constant values resulting from evaluating the level expression instead of referring to the level expression directly. The transformation technique is described in Appendix B. □

6.3 Rules Involving Selection and Generalized Projection

Selection and GP operations commute if the selection only refers to GROUP BY levels in the GP or to levels above them.

Example 6.7 $\Pi_{[Month,EC]<\text{SUM}(Cost)>}(\sigma_{Year=2000}(Purchases))$ is equivalent to
$\sigma_{Year=2000}(\Pi_{[Month,EC]<\text{SUM}(Cost)>}(Purchases))$ □

Rule 6.7 (Commutativity of Selection and Generalized Projection) The following holds if for each level L referenced in θ there exists a level $L' \in \mathcal{L}$ such that $L' \sqsubseteq L$:

$$\Pi_{[\mathcal{L}]<F(M)>}(\sigma_\theta(\mathcal{F})) \leftrightarrow \sigma_\theta(\Pi_{[\mathcal{L}]<F(M)>}(\mathcal{F}))$$

Reasoning: Since a single fact in the fact table will satisfy the selection predicate both before and after grouping, exactly the same facts will be selected by the selection operation in the two cases. □

Although a GP cannot always be pushed below a selection, part of it can if the selection predicate does not refer to measures. The way this is done is similar to Rule 6.4. The part that can be pushed below a selection must allow the predicate to be evaluated and it must also be possible to roll up to the levels specified in the original GP. In addition, the new GP may not roll up over a non-strict level relationship. This would prohibit further aggregation in the original GP.

Example 6.8 Consider again the new Purchases cube Purchases2 containing the dimension in Figure 7.

The expression $\Pi_{[Class,Country]<\text{SUM}(Cost)>}(\sigma_{NoOfPins>16}(Purchases2))$ is equivalent to $\Pi_{[Class,Country]<\text{SUM}(Cost)>}(\sigma_{NoOfPins>16}(\Pi_{[EC,Country]<\text{SUM}(Cost)>}(Purchases2))$. Here, the new GP must make it possible to roll up to both NoOfPins and Class. But since there is non-strictess between EC and Type it can not roll up to the Type level because this would prohibit further aggregation to the Class level. □

Rule 6.8 (Pushing Generalized Projection Below Selection) If a predicate θ does not contain references to any measures in M, then the following holds:

$$\Pi_{[\mathcal{L}]<F(M)>}(\sigma_\theta(\mathcal{F})) \leftrightarrow \Pi_{[\mathcal{L}]<F(M)>}(\sigma_\theta(\Pi_{[\mathcal{L}']<F(M)>}(\mathcal{F})))$$

where $\mathcal{L}' = \{L | \forall D \in \text{Dims}(\mathcal{L}, \theta)(L = \text{MaxStrict}(\{L' | \forall L'' \in \text{Levs}(D, \mathcal{L}, \theta)(L' \sqsubseteq_D L'')\})\}$.
Reasoning: If a dimension value in a fact aggregated by the new GP satisfies the selection predicate then all facts in the original cube that correspond to children of that dimension value will also satisfy the predicate. Notice that this is only true because the aggregation is over a strict hierarchy. □

The above rules do not allow selections to be interchanged with GPs if the predicates contain references to measures although it is possible to do this in some special cases. For the transformation to be legal the GP must not perform any aggregation. This is the case if all GROUP BY levels in the GP are already present in the cube or result from a decoration. If a level is created by decoration (without any GP rolling the decoration up in case of ALL decoration) then no measures are changed, since decoration adds a dimension without changing the number of facts.

Rule 6.9 (Commutativity of Generalized Projection and Selection with References to Measures) Let $\mathcal{L}$ and $\mathcal{L}'$ be sets of levels such that $\forall L \in (\mathcal{L} \cap \mathcal{L}')(L = xp_\perp$ for some decoration $\delta_{S[L_s, link, xp]})$ and M and M' be sets of measures such that $M \subseteq M'$. Then the following holds if for each level L referenced in θ there exists a level $L' \in \mathcal{L}$ such that $L' \sqsubseteq L$:

$$\Pi_{[\mathcal{L}]<F(M)>}(\sigma_\theta(\mathcal{F})) \leftrightarrow \sigma_\theta(\Pi_{[\mathcal{L}]<F(M)>}(\mathcal{F}))$$

where the schema for $\mathcal{F}$ is $\mathcal{L}', M'$.
Reasoning: Since the number of measure values do not change by applying the GP the same result is produced by evaluating the selection before and after the GP is applied. □

6.4 Rules Involving a Single Operator

Decoration operators commute if one operator does not decorate the other.

Rule 6.10 (Commutativity of Decorations) Let $L_1[S_1]/link_1/xp_1$ and $L_2[S_2]/link_2/xp_2$ be level expressions such that $L_1 \neq xp_2$. Then the following holds:

$$\delta_{S_1[L_1, link_1, xp_1]}(\delta_{S_2[L_2, link_2, xp_2]}(\mathcal{F})) \rightarrow \delta_{S_2[L_2, link_2, xp_2]}(\delta_{S_1[L_1, link_1, xp_1]}(\mathcal{F}))$$

Reasoning: Since dimensions in a cube are not ordered, the order of decorations is not important. □

A conjunctive selection can be split up in two selections and vice versa.

Rule 6.11 (Cascade of Selections) Let θ_1 and θ_2 be predicates. Then the following holds:

$$\sigma_{\theta_1 \wedge \theta_2}(\mathcal{F}) \leftrightarrow \sigma_{\theta_1}(\sigma_{\theta_2}(\mathcal{F}))$$

Reasoning: Selection only affects tuples in the fact table. Such a tuple satisfies the conjunctive predicate exactly when it satisfies the first predicate and then the second predicate. □

Selection operators commute.

Rule 6.12 (Commutativity of Selections) Let θ_1 and θ_2 be predicates. Then the following holds:

$$\sigma_{\theta_1}(\sigma_{\theta_2}(\mathcal{F})) \leftrightarrow \sigma_{\theta_2}(\sigma_{\theta_1}(\mathcal{F}))$$

Reasoning: Follows from Rule 6.11 and commutativity of conjunction. □

GPs can be split up if all GROUP BY levels and measures of the outer GP are also listed by the inner GP. In addition, if the lower GP only performs a partial aggregation in some dimensions this may not result in further aggregation being prohibited. Hence, the part of the dimension being aggregated over must be strict.

Rule 6.13 (Cascade of Generalized Projections) Let $\mathcal{L}$ and $\mathcal{L}'$ be sets of levels such that $\forall L \in \mathcal{L}(\exists L' \in \mathcal{L}'(L' \sqsubseteq L \wedge (L' \neq L \Rightarrow L' \sqsubseteq \text{MaxStrict}(\{L''|L'' \in \text{Dim}(L')\}))))$ and let $F(M)$ and $F(M)'$ be aggregate functions applied to measures such that $F(M) \subseteq F(M)'$. Then the following holds:

$$\Pi_{[\mathcal{L}]<F(M)>}(\Pi_{[\mathcal{L}']<F(M)'>}(\mathcal{F})) \leftrightarrow \Pi_{[\mathcal{L}]<F(M)>}(\mathcal{F})$$

Reasoning: This holds because aggregate functions are assumed to be distributive and dimensions are strict, i.e. summarizability is preserved. □

If a GP does not perform any aggregation on the cube to which it is applied, the GP can be removed.

Rule 6.14 (Redundant Generalized Projection) The following holds if the schema for $\mathcal{F}$ is $\mathcal{L}, M$:

$$\Pi_{[\mathcal{L}]<F(M)>}(\mathcal{F}) \leftrightarrow \mathcal{F}$$

Reasoning: No aggregation occurs in the GP since the same levels occur both before and after the GP is applied. Note that this is only true because the fact table does not contain duplicates. □

If a decoration is applied to an identical decoration one of them can be removed.

Rule 6.15 (Cascade of Decorations) Let $L[S]/link/xp$ be a level expression. Then the following holds:

$$\delta_{S[L,link,xp]}(\delta_{S[L,link,xp]}(\mathcal{F})) \leftrightarrow \delta_{S[L,link,xp]}(\mathcal{F})$$

Reasoning: This follows from Theorem 6.1. □

6.5 High Level Rules

The rules presented above can be combined to high level rules. This is demonstrated by presenting a derived rule which will be useful in Section 8.2.

Since a GP can be pushed below both decorations and selections, a single rule can do this across several such operations.

Rule 6.16 (Pushing Generalized Projection Below Decorations and Selections) Let $O_1, \ldots O_n$ be a set of selection and decoration operators. If the selections do not refer to measures, the following holds:

$$\Pi_{[\mathcal{L}]<F(M)>}(O_1(\ldots(O_n(\mathcal{F}))\ldots)) \leftrightarrow \Pi_{[\mathcal{L}]<F(M)>}(O_1(\ldots(O_n(\Pi_{[\mathcal{L}']<F(M)>}(\mathcal{F})))\ldots))$$

where $\mathcal{L}'$ is constructed by applying the definition of $\mathcal{L}'$ in Rule 6.4 or 6.8 consecutively for each O_i.
Reasoning: First, consider the left to right case. By applying Rule 6.4 and Rule 6.8 we get the following:

$$\Pi_{[\mathcal{L}]<F(M)>}(O_1(\ldots(O_n(\mathcal{F}))\ldots)) \rightarrow \Pi_{[\mathcal{L}]<F(M)>}(O_1(\Pi_{[\mathcal{L}_1]<F(M)>}(\ldots(O_n(\Pi_{[\mathcal{L}_n]<F(M)>}\mathcal{F}))\ldots)))$$

If we look at a sub-sequence: $\Pi_{[\mathcal{L}_{i-1}]<F(M)>}(O_i(\Pi_{[\mathcal{L}_i]<F(M)>}(\ldots)))$ there are two cases to consider:

$O_i = \delta_{S[L,link,xp]}$**:** Since $\mathcal{L}_i$ is constructed from Rule 6.4, the starting level is contained in $\Pi_{[\mathcal{L}_i]<F(M)>}$. Then it follows from Rule 6.3, that O_i and $\Pi_{[\mathcal{L}_i]<F(M)>}$ can be swapped.

$O_i = \sigma_\theta$**:** Then it follows from Rule 6.7, that O_i and $\Pi_{[\mathcal{L}_i]<F(M)>}$ can be swapped since θ does not refer to measures.

After O_i and $\Pi_{[\mathcal{L}_i]<F(M)>}$ has been swapped, we have the following sub-sequence: $\Pi_{[\mathcal{L}_{i-1}]<F(M)>}(\Pi_{[\mathcal{L}_i]<F(M)>}(O_i(\ldots)))$. Then it follows from Rule 6.13, that $\Pi_{[\mathcal{L}_i]<F(M)>}$ can be removed. If this procedure is repeated, starting from the sub-sequence $\Pi_{[\mathcal{L}_{n-2}]<F(M)>}(O_{n-1}(\Pi_{[\mathcal{L}_{n-1}]<F(M)>}(\ldots)))$ and up, then it follows that

$$\Pi_{[\mathcal{L}]<F(M)>}(O_1(\Pi_{[\mathcal{L}_1]<F(M)>}(\ldots(O_n(\Pi_{[\mathcal{L}_n]<F(M)>}\mathcal{F}))\ldots))) \rightarrow$$
$$\Pi_{[\mathcal{L}]<F(M)>}(O_1(\ldots(O_n(\Pi_{[\mathcal{L}']<F(M)>}(\mathcal{F})))\ldots))$$

Consider now the right to left case. Rule 6.3 always holds in the left to right direction, and Rule 6.7 holds in the right to left direction because no selections refer to measures. Thus, we get the following:

$$\Pi_{[\mathcal{L}]<F(M)>}(O_1(\ldots(O_n(\Pi_{[\mathcal{L}']<F(M)>}(\mathcal{F})))\ldots)) \rightarrow \Pi_{[\mathcal{L}]<F(M)>}(\Pi_{[\mathcal{L}'']<F(M)>}(O_1(\ldots(O_n(\mathcal{F})))\ldots))$$

According to Rule 6.13 the second GP can now be removed, giving:

$$\Pi_{[\mathcal{L}]<F(M)>}(\Pi_{[\mathcal{L}'']<F(M)>}(O_1(\ldots(O_n(\mathcal{F})))\ldots)) \rightarrow \Pi_{[\mathcal{L}]<F(M)>}(O_1(\ldots(O_n(\mathcal{F})))\ldots))$$

$\square$

The usefulness of this and the remaining rules presented in Table 2 will become clear in Section 8. Before that, an overview of the federation architecture is given.

7 Federation Architecture

This section describes the architectural design of a prototype system supporting the SQL$_{XM}$ query language. The system allows enumerated and natural links to be defined and used in SQL$_{XM}$ queries for decoration, selection, and grouping. Three different semantics, ANY, ALL, and CONCAT, can be specified when using the links, providing a flexible way to handle different cardinalities between dimension values and XML nodes. The following sections (8-11) describe different aspects of the evaluation and optimization of queries in the system, while Section 12 discusses implementation aspects and experimental studies.

Generally, current OLAP systems do not allow non-strict dimensions which is necessary to provide flexible access to external data. Furthermore, creating dimensions is often an expensive process requiring the cube to be rebuilt, which makes it unfeasible to do this for each query. Some OLAP systems, such as MS Analysis Services [TSC99], allow so-called *changing dimensions* to be created, which do not require the cube to be fully processed. However, a partial processing is still needed, making it unfeasible to do at query time. Furthermore, non-strict dimensions are not allowed. Consequently, a different approach is taken here, that allows any OLAP system to be used. The basic idea is to evaluate a SQL$_{XM}$ query by constructing and evaluating the OLAP and XML queries separately, and combine the results of these queries using a temporary component. The architecture is shown in Figure 8.

The key component is the Federation Manager, which processes SQL$_{XM}$ queries fed to it by the user interface by fetching data from the OLAP and XML components. Intermediate interface components are inserted between the Federation Manager and the OLAP and XML components to make the Federation Manager independent of the query languages used by these components. The Federation Manager uses three auxiliary components to store meta data, link data, and temporary data used in the evaluation of a SQL$_{XM}$ query, respectively. The meta data component contains descriptions of the dimensions in the OLAP component, whereas the

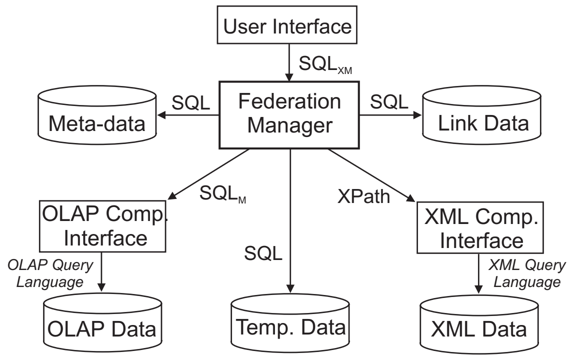


Figure 8: Overall architecture of the prototype supporting the SQL_{XM} query language.

link data component contains link specifications as described in Section 4. The temporary data component is used for storing intermediate results during the processing of a query. All three auxiliary components assume only an SQL interface. Whenever possible these components, in particular the temporary component, should be placed on the same host as the OLAP component to minimize data transportation costs. However, this may not always be possible if e.g. the federation and the OLAP component are managed by separate departments.

A relational DBMS is used for the temporary component because the final result can be computed by joining the fact table resulting from the OLAP component query, and tables representing decoration data. As stated in Definition 4.4 the result of a level expression is a set of (dimension value, decoration value) pairs, which is easily represented in a table. Implementing the different semantics of level expressions is straightforward, as this only differs in which values are inserted into the table. Also, the special N/A value is easily handled by using the left outer join operator when joining the fact table and the tables containing the decoration data. Since NULL values are not legal dimension or decoration values, the NULL values introduced by the outer join can be treated as N/A values without confusion.

A prototype based on the architecture in Figure 8 is currently being developed, which is described further in Section 12. To give an overview of how a SQL_{XM} query is evaluated in this architecture, we now present a simple example showing the basic steps. A more detailed description of the evaluation process as well as a discussion of heuristic optimization is presented in Section 8, while Section 9 discusses cost based optimization techniques.

Example 7.1 A user poses the following query to the *Federation Manager*, which decorates Supplier with SName:

```
SELECT     SUM(Cost), Supplier, Class(EC), Supplier/Sup_Link/SName
FROM       Purchases
GROUP BY   Supplier, Class(EC), Supplier/Sup_Link/SName
```

As mentioned, the basic idea in evaluating such a query is to fetch data from the OLAP and XML components, and then combine the results using the temporary component. The given SQL_{XM} query is analyzed and component queries are constructed. In this case, the following OLAP component query is posed:

```
SELECT     SUM(Cost), Supplier, Class(EC)
FROM       Purchases
GROUP BY   Supplier, Class(EC)
```

Note that only two dimensions are needed here, because the level being decorated (Supplier) must also be present in the result. This would e.g. not be the case if the Month level was decorated instead. The fact table resulting from this OLAP query is stored in the temporary component, and shown here:

Cost	Supplier	Class
2940	S1	FF
6900	S3	FF
32050	S2	FF
9480	S3	L

We will refer to this table as "TempFactTable" in the following.

To fetch the data from the XML components, the given SQL$_{XM}$ query is analyzed, and all unique level expressions are identified. For each of these level expressions, XML component queries are constructed, and a table representing the resulting (dimension value, decoration value) pairs is created in the temporary component. In this case there is only one level expression: Supplier/Sup_Link/SName. Based on the definition of Sup_Link in Example 4.1, the following component queries are constructed: /Components/Supplier[@SCode='SU13']/SName and /Components/Supplier[@SCode='SU15']/SName. The first of these results in the decoration values for $S1$, whereas the second one results in the decoration values for $S3$, according to the definition of Sup_Link. Since no semantic modifier is given in the level expression, ANY semantics is assumed. Hence, one of the resulting nodes of the first component query is paired with $S1$, and so on. This results in the following table being added to the temporary component:

Supplier	SName
S1	John's ECs
S3	Jane's ECs

We will refer to this table as "Supplier_SName" in the following.

The final step is to construct and evaluate a plain SQL query in the temporary component combining the data retrieved from the OLAP and XML components. In this case, no further selection or grouping is required, but in general this can be necessary. The component data is combined using a left outer join operation, resulting in the following SQL query being posed to the temporary component:

```
SELECT   SUM(Cost) AS Cost, Supplier, Class, SName
FROM     TempFactTable NATURAL LEFT OUTER JOIN Supplier_SName
```

This query results in the following table, which is the resulting fact table of the SQL$_{XM}$ query:

Cost	Supplier	Class	SName
2940	S1	FF	John's ECs
6900	S3	FF	Jane's ECs
32050	S2	FF	NULL
9480	S3	L	Jane's ECs

Note that by using the left outer join operator, a NULL value is added where no decoration value is available. These NULL values can be treated as the special N/A value because no other NULL values occur as dimension or decoration values. □

8 Query Evaluation

The processing of a SQL$_{XM}$ query can be divided into three main tasks: Constructing and evaluating component queries, retrieving and storing temporary data resulting from these queries, and processing temporary data needed to produce the final result. The most significant way to improve the total evaluation time of a federation query will generally be to reduce the amount of temporary data. The benefits of this are several: It reduces data transfer costs, the time it takes to store temporary data, and the time required to produce the final result. Also, it will usually reduce the combined query processing time as the OLAP component generally performs multidimensional queries faster than a relational DBMS. The transfer costs will be particularly significant if temporary data is stored outside the OLAP component. Hence, a primary goal of our optimization efforts lies in reducing the amount of temporary data required to evaluate a SQL_{XM} query. Because OLAP systems typically contain large amounts of data, this is particularly important for the OLAP component. Thus, queries are evaluated by investigating how to split a SQL_{XM} query tree into a part that can be evaluated entirely in the OLAP component, and one that must be evaluated in the relational component because it involves XML data. We refer to this splitting process as *partitioning* the query tree. This partitioning is based on the transformation rules described in Section 6 and will be discussed in detail in Section 8.2. The partitioned query tree is used to create the component queries, as is discussed in Section 8.3. First, we present the architecture of the *Federation Manager* and the steps involved in evaluating a SQL_{XM} query.

8.1 Architectural Design of the Federation Manager

The architecture of the Federation Manager and a single Component Interface is shown in Figure 9.

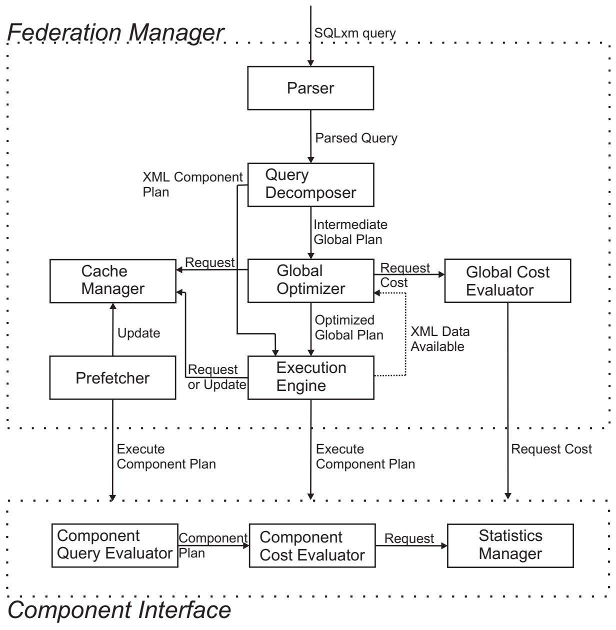


Figure 9: Architecture of the Federation Manager and a Component Interface.

When a SQL_{XM} query is posed to the Federation Manager, it is parsed and transformed into a query tree as described in Section 5.4. The next step in evaluating a SQL_{XM} query is to transform, or partition, the initial query tree into a form, where redundant operators are removed, and from which component queries can easily be formed. The result of this is an OLAP component plan and a temporary component plan, jointly referred to as a *Global Plan*. This task is handled by the *Query Decomposer*, and is described in detail in Section 8.2. After eliminating redundant decoration nodes, the retrieval of XML data can be started. Hence, the *Query Decomposer* dispatches a series of XML Component Plans to the *Execution Engine* which fetches the XML data. In parallel with this, it invokes the *Global Optimizer* by passing on the global plan.

The *Global Optimizer* generates a number of different global plans by considering the use of cached intermediate results, and by considering whether or not to inline XML data into the OLAP component query. It then chooses a global plan by considering the cost of each generated plan, and dispatches this plan to the *Execution Engine*. Estimating the cost of a global plan is handled by the *Global Cost Evaluator*, which determines the global cost by requesting cost information from each component interface. Cost based optimization and the estimation of cost information is described in detail in Section 9 and 10, respectively.

The *Execution Engine* handles the execution of component queries. It either receives an XML component plan, which is forwarded to the relevant component interface, or it receives a global query plan. From a global plan the *Execution Engine* forms an OLAP component query, and a temporary component query. It executes the OLAP query to the OLAP component interface, and when the results of this query and the XML component queries are available in the temporary component, it executes the temporary component query, and returns the result. If the result of an XML query becomes available before it was anticipated and the OLAP query has not yet been posed, the global plan is reconsidered. (Thus, a *continuously adaptive* [AH00] approach is used.) The construction of component queries is described in detail in Section 8.3. Upon completion of the component queries, the *Execution Engine* informs the *Cache Manager* about the intermediate results, that have been added to the temporary component, during execution of the SQL_{XM} query.

The Federation Manager also uses pre-fetching of intermediate results to increase query performance. This is handled by the *Pre-fetcher*. When the load of the system is low, the *Pre-fetcher* executes a number of component queries, and stores these intermediate results in the temporary component. It then informs the *Cache Manager* about these results, making them available for use in subsequent SQL_{XM} queries. Pre-fetching and caching is discussed further in Section 9.4.

Each Component Interface comprises a *Component Query Evaluator*, a *Component Cost Evaluator*, and a *Statistics Manager*. By using the *Component Cost Evaluator*, the Component Interface is able to perform some cost based local query optimization before posing queries to the component. The *Statistics Manager* obtains and maintains statistical information about the component. Obtaining statistical information can be done in a number of different ways, one of which is by using so-called *probing queries*. This technique is discussed in Section 10.

After presenting the query evaluation and optimization techniques mentioned here, an overview of their combined use in the federated system is provided in Section 11.

8.2 Partitioning Federation Queries

The main problem when considering global optimization of SQL_{XM} queries is how to determine which part of a query can be evaluated in the OLAP component and which cannot. Let us for now assume that only the part that does not refer to XML data can be evaluated in the OLAP component. In Section 9.2 we discuss how to integrate XML data into the OLAP query, reducing the amount of data produced by the OLAP component. This assumption leaves the problem of splitting, or *partitioning*, the query tree described in Section 5.4 in two parts, such that as much as possible of the query is evaluated in the OLAP component.

In Figure 10 the entire partitioning process of a query tree is shown. In the figure, Δ refers to a *sequence* of decorations, and Σ to a *sequence* of selections. Figure 10(a) represents the initial form of the query tree, constructed as described in Section 5.4, whereas Figure 10(g) represents the general result of the partitioning. The figures 10(b)-(f) represent intermediate steps of the partitioning algorithm, and will be discussed in detail later. All operations from the bottom decoration and upward must be evaluated in the relational component since it refers to XML data. Only the part below the bottom decoration can be evaluated in the OLAP component. Thus, Figure 10(a) represents the query that retrieves the entire cube and evaluates the entire query in the temporary component. To avoid this, we partition the query tree to the form shown in Figure 10(g) in such a way, that the OLAP query will aggregate as much as possible, while still allowing the decorations to be performed.

Example 8.1 In this example we assume that the Class level is linked to an AverageMonthlyPurchases node and that the Year level is linked to an ExpectedPurchases node. These nodes are not part of the Components document, but a budget.xml document. The structure of this document is indicated by the regular expression:
$\mathsf{Budget(Class(@ClassCode,\ AverageMonthlyPurchases^{+},Year(@YearCode,\ ExpectedPurchases^{+})^{+})^{+})}$

Consider the partitioning of the query in Figure 11. Here, the selection $Year > 1995$ and most of the aggregation can be evaluated in the OLAP component, since it does not refer to XML data. The partitioned query is shown in Figure 11(g). Again, the figures 11(b)-(f) represents intermediate steps and will be discussed in detail later. $\square$

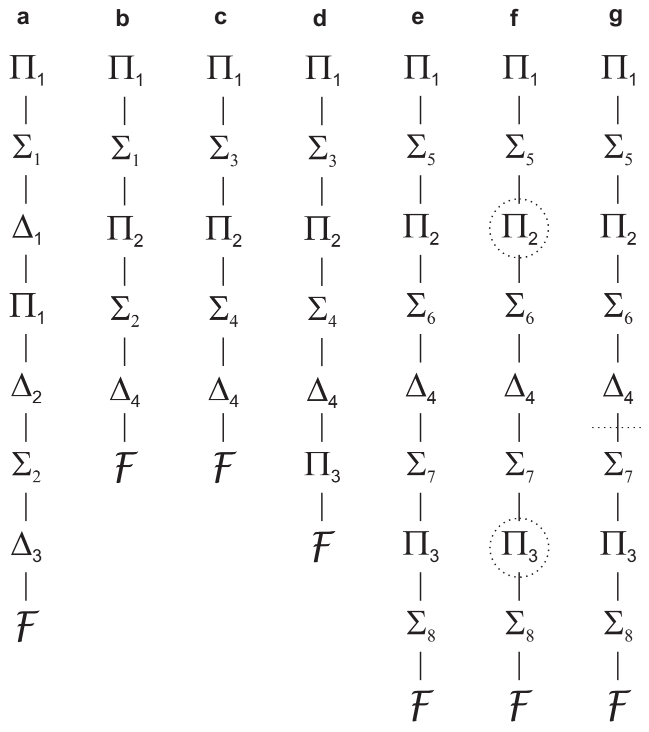


Figure 10: Partitioning of a query.

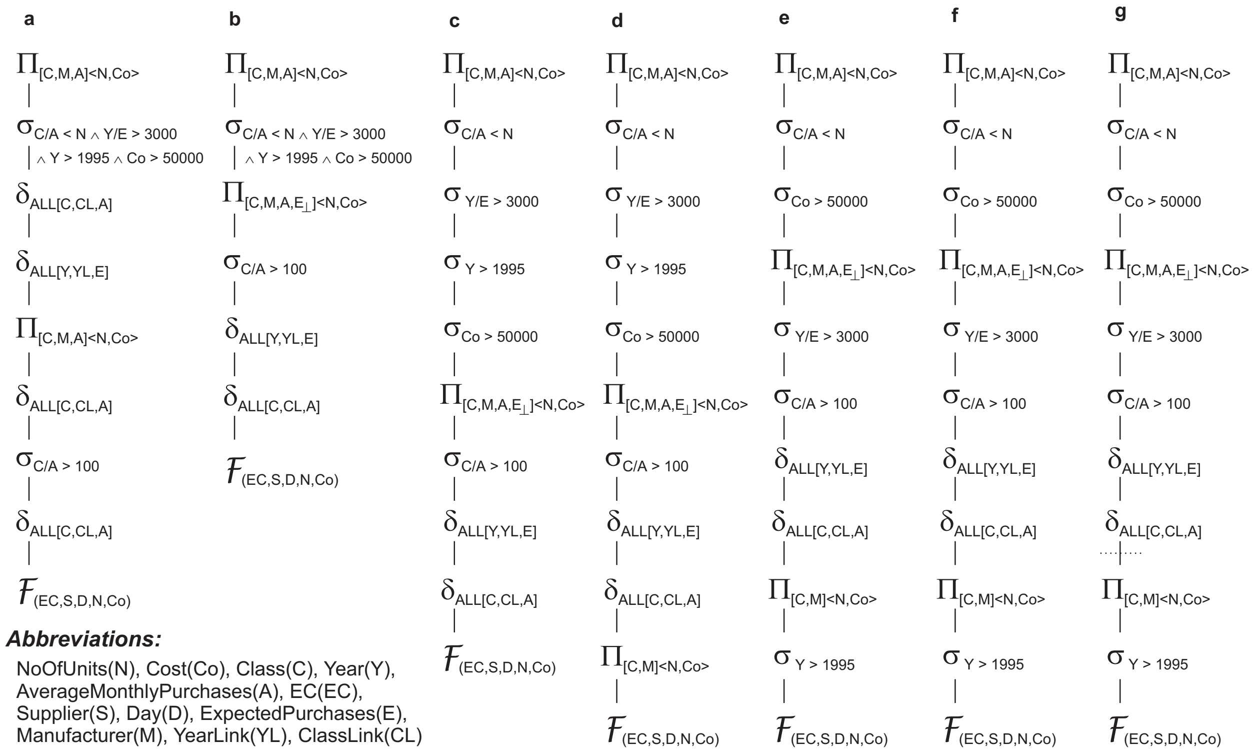

Abbreviations:
NoOfUnits(N), Cost(Co), Class(C), Year(Y), AverageMonthlyPurchases(A), EC(EC), Supplier(S), Day(D), ExpectedPurchases(E), Manufacturer(M), YearLink(YL), ClassLink(CL)

Figure 11: Example of query partitioning.

Although the heuristic approach of pushing as much towards the component as possible is often used, it is not generally valid, as pointed out by [ROH99]. The reason for this, is that if joins are used, it may sometimes result in more data being transferred between components and the federation manager. However, the heuristic is valid in this case since the operations considered always reduce the size of the result. Hence, letting the OLAP component perform as much of the evaluation as possible is optimal in this setting.

From the partitioned query tree an OLAP query, a number of XML queries, and a relational query working on the temporary results are constructed. In many cases, the result of the OLAP query will be comparable in size to the result of the total query. This is true in the presumably common case where the referenced XML data is linked to values that are already present in the result, such as in Example 8.1. Certain exceptional queries, such as comparing each measure value in the cube to XML data in the WHERE clause, can, of course, not generally be evaluated efficiently, since the entire cube would have to be transferred to the temporary component. However, this is not likely to be needed in practice. Notice that if there is no HAVING clause in the query, only the bottom GP is present in the query tree. Also, note that the two GPs are always identical, since the only purpose of the top GP is to remove any decorations introduced above the bottom GP. Each of the following steps are explained in general terms by referring to Figure 10, and the same steps are demonstrated on the Example in Figure 11.

The partitioning is performed by applying the transformation rules in Section 6 directly or by applying higher level rules derived from them. The partitioned query tree should satisfy two requirements: First, as much of selections and GPs as is possible, by using the transformation rules, must occur below the bottom decoration. Second, it should have a form, such that it is easy to construct the component queries. The following sections present algorithms, which perform such a transformation, beginning with the overall algorithm.

8.2.1 The Overall Partitioning Algorithm

Algorithm 8.2.1 transforms a query tree on the form shown in Figure 10(a) to an equivalent query tree on the form shown in Figure 10(g). We consider the non-trivial case where at least one decoration occurs in the query tree. Pure OLAP queries issued to the federation can be identified at parse-time and passed on to the OLAP component.

Algorithm 8.2.1 Partitioning a Query Tree

```
1 PartitionQueryTree(RootNode)
2   RemoveAndPushDecorations(RootNode, $\emptyset$)
3   SplitSelections(RootNode)
4   LowerGP := FindLowerGP(RootNode)
5   PushGPDown(LowerGP.Child, LowerGP)
6   PushSelectionsDown(RootNode, $\emptyset$)
7   RemoveRedundantGPs(RootNode)
```

The algorithm begins by removing redundant decorations and pushing them as far down the query tree as possible. After this, selection nodes are split allowing them to be moved more freely. Then it is identified to which levels the cube can be aggregated by pushing the lower GP down, followed by identifying which selections can be made in the cube by pushing down selections. Finally, any redundant GPs are removed from the query tree. Each function mentioned in lines 2-3 and 5-7 refers to the Algorithms 8.2.2-8.2.6 which are discussed in the following sections. All of these algorithms only visit each node once. Hence, five traversals are performed during partitioning. Although explained separately, these algorithms can to some extent be combined such that fewer traversals are needed.

8.2.2 Removing and Pushing Decorations Down

Two important observations can be made about the occurrence of decorations: First, only a single decoration is necessary to evaluate a predicate which refers to decoration data. Second, the number of decorations (zero or one) occuring as GROUP BY levels in a GP is only important in the top GP. For other GPs, decorations can be removed or added to improve efficiency. Hence, only the lowest occurrence of a particular decoration is needed. Because decorations are implemented as joins, it is generally cheaper to perform a single decoration rather than decorating, removing the decoration, and decorating again. Thus, the number of identical decorations should be reduced to one as shown on Figure 10(b). This is generally possible by using the rules 6.1, 6.3, 6.5, and 6.10.

Example 8.2 Only the two lowest decorations are needed in Figure 11(a). The top occurrence of $\delta_{\text{ALL}[C,CL,A]}$ can be removed using rules 6.1 and 6.10. The middle occurrence can be removed using Rule 6.5 and Rule 6.15. The decoration $\delta_{\text{ALL}[Y,YL,E]}$ can be moved down because of Rule 6.3 and Rule 6.5. □

Algorithm 8.2.2 removes all redundant decorations, and pushes all remaining decorations down to the base cube. As it will become clear later, this makes it easier to identify to which levels it is possible to aggregate the cube, as well as to identify which selections can be performed in the OLAP component.

Algorithm 8.2.2 Remove and Push Decorations

```
 1 RemoveAndPushDecorations(Node, Decorations)
 2   Case node of:
 3     $\delta$ :
 4       Decorations := Decorations $\cup$ {Node}
 5       remove Node from query tree
 6     $\Pi$ :
 7       for all n $\in$ Decorations
 8         if Rule 6.1 applies
 9           remove n from Decorations
10         else // Rule 6.3 applies
11           change Node according to Rule 6.3
12     $\sigma$ :
13       // Rule 6.5 always applies. Do nothing.
14     Base :
15       for all n $\in$ Decorations
16         insert n above Node
17       return
18   RemoveAndPushDecorations(Node.Child, Decorations)
```

The algorithm for removing and pushing decorations is applied recursively on the tree beginning from the root node. In addition to the current node, it takes a set of decorations as argument, in which decorations can be stored temporarily. It follows from Rule 6.3 and 6.5, that decorations can always be pushed down the query tree to the base cube. Hence, every time the algorithm sees a decoration it is kept for later reference (lines 3-5), and finally inserted above the base cube (lines 14-17). Since Decorations is a set, i.e. contains no duplicates, redundant decorations are removed in lines 4-5 in accordance with Rule 6.15. When the algorithm encounters a generalized projection it goes through Decorations and for each decoration determines whether it can be considered redundant (line 8) or whether it should be pushed down below the generalized projection (lines 10-11). Note that Rule 6.3 always applies, as long as the generalized projection is changed accordingly (line 11). According to Rule 6.5, a decoration can always be pushed below a selection node. Hence, whenever the algorithm encounters a selection node (lines 12-13) Node and Decorations are left unchanged.

8.2.3 Splitting Selections

Conjunctive selections are split using Rule 6.11 to be able to move them around more freely. Hence, a number of new selections are present in Figure 10(c).

Example 8.3 The splitting of selections is shown in Figure 11(c). □

Algorithm 8.2.3 Split Selections

```
1  SplitSelections(Node)
2    Case Node of:
3      $\sigma$ :
4        ListOfPredicates := SplitPredicate(Node.Predicate)
5        LastNode := Node.Parent
6        for all p' $\in$ ListOfPredicates
7          create new $\sigma_{p'}$ with predicate p'
8          make $\sigma_{p'}$ child of LastNode
9          LastNode := $\sigma_{p'}$
10       make Node.Child child of LastNode
11     Base :
12       return
13   SplitSelections(Node.Child)
```

Algorithm 8.2.3 splits selection operator nodes recursively from the top node and down.

The algorithm for splitting selections takes a Node as input. Based on the type of this node (line 2) the algorithm either returns (line 13) in case it is a base cube, or it splits one selection node into several selection nodes (lines 3-10). The splitting is based on the function SplitPredicate, which splits a conjunctive predicate into its subparts. Each new predicate is used to create a new selection node, which is added to the query tree just below the parent of the original node (lines 7-10). According to Rule 6.11, the original query tree is equivalent to the query tree obtained by applying Algorithm 8.2.3 to the root.

8.2.4 Pushing Generalized Projections Down

At this point in the partitioning algorithm, the query tree is on the form shown in Figure 10(c). That is, all decorations are at the bottom of the query tree just above the base federation. The next step is to push the lower of the two generalized projections below all the decorations. It follows from Rule 6.16, that a GP can be pushed below a number of selections and decorations. The result of this rule is the addition of a single GP at the bottom of the query tree as shown in Figure 10(d). It is only necessary to use this procedure on the second GP, since the two original GPs are identical except for any decorations introduced when moving decorations as described in Algorithm 8.2.2.

Example 8.4 The result of using Rule 6.16 on the example is shown in Figure 11(d). It can easily be seen that the cube produced by the new GP is sufficient for applying the decorations and evaluating the selections above it. □

Algorithm 8.2.4 identifies to which levels it is possible to aggregate the cube, while still allowing the above decorations and selections to be made.

This algorithm is an implementation of Rule 6.16, and is again called recursively. The input GP is the generalized projection which should be inserted in the query tree just above the base cube. It is modified each time it is pushed below a decoration (line 9) or a selection (line 5). If the generalized projection cannot be pushed down to the base cube (lines 4-7), then no generalized projection is inserted at all. In that case, the only way to compute the final result is by fetching all data from the cube, and then combine this with the external data in the temporary component. Since Algorithm 8.2.4 should be called on the child of the lower GP, it is not nessesary to handle the case where Node is a GP.

8.2.5 Pushing Selections Down

After identifying to which levels the cube can be aggregated, the amount of data fetched from the cube is further reduced by pushing selections down the query tree. Selections can be pushed down using the rules 6.5, 6.7, 6.9,

Algorithm 8.2.4 Push Generalized Projection Down.

```
PushGPDown(Node, GP)
  Case Node of:
    $\sigma$ :
      if $\theta$ does not refer to measures
        change GP according to Rule 6.8
      else
        return // GP is not inserted in the query tree
    $\delta$ :
      change GP according to Rule 6.4
    Base :
      insert GP above Node
      return
  PushGPDown(Node.Child, GP)
```

and 6.12 thereby introducing selections below the bottom decoration. The selections that can be pushed below the bottom decoration are those that do not refer to decorations or measures. An exception from this is that selections containing references to measures but not decorations can be pushed down, if the entire GP was pushed down to the bottom in the previous step. In that case, the measure values will be the same below the bottom decoration but above the bottom GP. Hence, selections can occur both below and above the bottom GP as shown in Figure 10(e). If the entire GP was pushed down, selections referring to both measures and decorations can also be pushed below the middle GP, but not below the decorations. Thus, Σ_5 will be empty. Notice that selections that refer to decorations are placed in a single group above the decorations, as this eases the construction of component queries.

Example 8.5 The result of pushing selections down is shown in Figure 11(e). Neither of the selections that refer to measures can be pushed down to the bottom, since the bottom GP is different from the middle GP. The selection that refers only to levels can be pushed down to the base cube, while the one that refers only to a decoration can be moved down to the decorations. □

Algorithm 8.2.5 pushes selection nodes as far down the query tree as possible.

The algorithm again traverses the query tree recursively. Whenever the input node Node is a selection node, it is removed from the query tree and kept in Selections (lines 3-5), until it can be reinserted. The reinsertion of selection nodes is done in four locations in the query tree, the first of which is right above the middle GP (lines 16-18). The selection nodes inserted here, are those referring to measure values after grouping has been done. Note that no nodes are inserted when the algorithm is invoked with the upper GP as an argument. This is due to the fact, that at this point no selections have been removed from the query tree and placed in the set of Selections. The second location in which selection nodes are inserted into the query tree is right above the uppermost decoration node. This is done whenever the algorithm is invoked with this decoration node as an argument (lines 7-10). All selection nodes placed in the set of Selections, which refer to decoration data, are inserted and removed from Selections. Hence, all selection nodes referring to decoration data are inserted here, leaving all such nodes in one place in the query tree. Selection nodes which can be pushed below decorations, but cannot be pushed below the lower GP, i.e., those nodes to which the rules mentioned in lines 13-14 do not apply, are inserted right above the lower GP. Finally, the remaining selection nodes in Selections are reinserted right above the base cube (lines 19-21). Selections inserted here are those to which the rules mentioned in lines 13-14 did apply, i.e. those that could be pushed below the lower GP.

8.2.6 Removing Redundant Generalized Projections

In special cases, the two GPs indicated in Figure 10(f) can be removed after having pushed selections down. First, if all selections in Σ_5 has been pushed down, the middle GP can be removed. This is the case if the entire

Algorithm 8.2.5 Push Selections Down

```
PushSelectionsDown(Node, Selections)
  Case Node of:
    $\sigma$ :
      add Node to Selections
      remove Node from query tree
    $\delta$ :
      for all n in Selections
        if Node.RefersToDecoration
          insert n above Node
           remove n from Selections
     $\Pi$ :
       for all n in Selections
         if Rule 6.7 applies or
           Rule 6.9 applies or
           // nodes can be swapped
         else
           insert n above Node
           remove n from Selections
     Base :
       for all n in Selections
         insert n above Node in query tree
       return
   PushSelectionsDown(Node.Child, Selections)
```

GP was pushed down to the bottom in Step 4. Second, if the GP that was pushed down to the bottom does not perform any aggregation of the input federation, it can be removed. This is equivalent to not being able to push the middle GP down and means that no aggregation can be performed in the OLAP component. The rules used to remove the GPs are 6.13 and 6.14, respectively.

Example 8.6 None of the GPs can be removed in the example as indicated by Figure 11(f). □

Algorithm 8.2.6 checks whether either of the two rules can be applied, and removes redundant GPs accordingly.

Algorithm 8.2.6 Remove Redundant Generalized Projections

```
RemoveRedundantGPs(RootNode)
  locate $\Pi_{Top}$, $\Pi_{middle}$ and $\Pi_{Bottom}$
  if $\Pi_{Top}$ is equal to $\Pi_{middle}$
    // Rule 6.13 applies
    remove $\Pi_{middle}$ from query tree
  if $\Pi_{Bottom}$ is equal to base cube
    // Rule 6.14 applies
    remove $\Pi_{Bottom}$ from query tree
```

After the removal of redundant GPs the query tree can be partitioned into two parts. Everything below the bottom decoration can be evaluated in the OLAP component, while the rest involves XML data and must be evaluated in the temporary component. Hence, the query tree can be split as indicated in Figure 10(g). Notice how the resulting query tree has a structure that makes it suitable for translation to component queries. This is described in the next section.

Example 8.7 The partitioned query tree is shown in Figure 11(g). □

8.3 Constructing Component Queries

After applying Algorithm 8.2.1 on the query tree, component queries can be constructed directly from the tree. Specifically, it can be divided into two parts as indicated in Figure 10(g): The bottom part, which can be translated to SQL_M, and the upper part, which can be translated to a plain SQL statement. The formulation of XML component queries is based on the decoration operators. In the following, we describe the component query construction for each of the components.

8.3.1 Constructing the OLAP Query

The bottom part of the query tree in Figure 10(g) can be refined to: $\sigma_{\theta_7}(\Pi_{[\mathcal{L}_3]<F(M_3)>}(\sigma_{\theta_8}(\mathcal{F})))$, where all selections in each of the two Σ blocks have been combined to a single conjunctive selection.

From this a SQL_M query is constructed:

```
SELECT      $F(M_3), \mathcal{L}_3$
FROM        $\mathcal{F}$
WHERE       $\theta_8$
GROUP BY    $\mathcal{L}_3$
HAVING      $\theta_7$
```

where the predicates and levels are converted to the syntax of SQL_M.

Example 8.8 From the bottom part of the query tree in Figure 11 the following SQL_M query is constructed:

```
SELECT      SUM(Cost), SUM(NoOfUnits), Class(EC), Month(Day)
FROM        Purchases
WHERE       Year(Day) > 1995
GROUP BY    Class(EC), Month(Day)
```

The fact table of the resulting cube is referred to as "Purchases$'$" in the following examples. ☐

8.3.2 Constructing XML Queries

As mentioned in Section 7 each decoration operator results in a table of (dimension value, decoration value) pairs being added to the temporary component. The XML queries that are needed to fetch the relevant data from the XML components, depend on the type of link used in the level expression, and the query interface offered by the XML component. For enumerated links, all dimension values may refer to different parts of an XML document, possibly even in different documents. Hence, because of the high degree of flexibility offered by enumerated links, in the worst case one XML query is needed per tuple in the enumerated link. Assuming that an XPath query interface is available, then for each tuple (e, URI, locator) in the enumerated link, the following XPath query is formed: locator/xp, where xp is the XPath expression from the decoration operator.

For natural links, a node is identified for each dimension value by the locator part of the link specification. Thus, an XML query must relate the node identified by the locator to the node identified by the user specified XPath expression. Since XPath does not allow queries such as (base/locator, base/xp), a more powerful language is needed to express this in a single query. Here, the query is given in XQuery [W3C01b], the current W3C working draft for an XML query language. Base, locator and xp represent XPath expressions from the level expression, and $v_1, \ldots, v_k$ are the dimension values from the starting level or, possibly, the alias values:

```
<Result>
LET $dimvalues = [<val>$v_1$</val>...<val>$v_k$</val>]
FOR $b IN base
  FOR $l IN $b/locator
  WHERE SOME $v IN $dimvalues SATISFIES $v.data() = $l.data()
  RETURN
    FOR $x IN $b/xp
    RETURN
      <ResultPair>$l, $x<ResultPair>
</Result>
```

For all base/locator nodes equal to one of the dimension values, a set of nodes is constructed. Each of these nodes is a pair of the locator and all decoration nodes returned by base/xp. If a large part of the base nodes are retrieved using this technique, it will often be faster to fetch *all* the (locator, xp) pairs. This decision is made from the estimated cost of performing each of these queries as is discussed further in Section 9.3 and 10.2.

If XPath is the only language available, then the general approach is to construct a query for each dimension value in the starting level by combining the base and locator parts with the user specified XPath expression. Thus, a query base[locator=e]/xp is constructed for each dimension value e. Contrary to the use of enumerated links, an extra meta data query is needed to retrieve the dimension values when using natural links.

Example 8.9 From the decoration operator $\delta_{ALL[Year,YearLink,ExpectedPurchases]}$, and the natural link (“Year”, “www.comp-org.org/budget.xml”, “/Budget/Class/Year”, “@YearCode”), the following XPath expressions are formed:

/Budget/Class/Year[@YearCode=’2000’]/ExpectedPurchases
/Budget/Class/Year[@YearCode=’2001’]/ExpectedPurchases
/Budget/Class/Year[@YearCode=’2002’]/ExpectedPurchases □

Posing many XPath expressions will often be computationally expensive, but this can be avoided as is discussed in Section 9.3.

8.3.3 Constructing the Relational Query

Let the result of the SQL$_M$ query be $\mathcal{F}'$, and $T_{\delta_1}, \ldots, T_{\delta_k}$ be relational tables that represent the results of the XML component queries. Each T_{δ_i} has two columns, one for the starting level of δ_i and one for the corresponding decoration values. Similarly, tables $T_{RU_1}, \ldots, T_{RU_l}$ are stored for each roll-up level mentioned in the query, where each T_{RU_i} has one column for a bottom level in $\mathcal{F}'$ and one for the corresponding roll-up level.

The upper part of the query tree in Figure 10(g) can be refined to:

$$\Pi_{[\mathcal{L}_1]<F(M)>}(\sigma_{\theta_5}(\Pi_{[\mathcal{L}_2]<F(M)>}(\sigma_{\theta_6}(\delta_{S_1[L_1,link_1,xp_1]}(\ldots(\delta_{S_k[L_k,link_k,xp_k]}(\mathcal{F}'))))$$

where all selections in Σ_5 and Σ_6 are collapsed into σ_5 and σ_6, respectively.

From this a plain SQL query is constructed, based on the fact table F' of the federation $\mathcal{F}'$:

```
SELECT   DISTINCT $M, \mathcal{L}_1$
FROM     (SELECT    $F(M)$, $\mathcal{L}_3$
          FROM      $F'$ NATURAL JOIN $T_{RU_1}$
                       ... NATURAL JOIN $T_{RU_l}$
                       NATURAL LEFT OUTER JOIN $T_{\delta_1}$
                       ... NATURAL LEFT OUTER JOIN $T_{\delta_k}$
          WHERE     $\theta_6$
          GROUP BY  $\mathcal{L}_3$
          HAVING    $\theta_5$)
```

where the predicates are converted to SQL syntax and $\mathcal{L}_3 = \mathcal{L}_1 \cup \{RU_1, \ldots, RU_l\} \cup \{\delta_1, \ldots, \delta_k\}$.

Notice that the roll-up and decoration columns are not removed by the aggregation, but is instead removed by projection in the outer SELECT statement. This is necessary because otherwise duplicated facts introduced by non-strict roll-up or decoration would be aggregated, producing a wrong result. Also, these may be needed to evaluate the HAVING clause.

Example 8.10 From the top part of the query tree in Figure 11 the following SQL query is constructed:

```
SELECT   DISTINCT Cost, NoOfUnits, Class, Month, AverageMonthlyPurchases
FROM     (SELECT    SUM(Cost) AS Cost, SUM(NoOfUnits) AS NoOfUnits, Class, Month, Year,
                    AverageMonthlyPurchases, ExpectedPurchases
          FROM      Purchases' NATURAL JOIN Month_Year
                    NATURAL LEFT OUTER JOIN Year_ExpectedPurchases
                    NATURAL LEFT OUTER JOIN Class_AverageMonthlyPurchases
          WHERE     AverageMonthlyPurchases > 100 AND ExpectedPurchases > 3000
          GROUP BY  Class, Month, Year, AverageMonthlyPurchases, ExpectedPurchases
          HAVING    SUM(Cost) > 50000 AND AverageMonthlyPurchases < SUM(NoOfUnits))
```
□

Although the rule-based partitioning is sufficient to achieve good performance for the most common types of queries, other queries may be expensive. This is true if none or little of the GP can be pushed below the bottom decoration. However, in the next section cost based techniques are presented to optimize such queries significantly.

9 Optimization Techniques

In this section we present a number of cost based optimization techniques that have been applied in the federated system. As noted by [SL90] the high degree of flexibility offered by federated systems comes at the cost of more difficult query optimization. The main reason for this is the lack of knowledge about the query processing abilities of component data sources. Here, this is especially true for the XML components as they will often reside on the Internet, where few or no assumptions can be made about the underlying data source and how component queries are optimized. For instance, it may be difficult or impossible to determine important factors like which access paths are available, which algorithms can be used to execute operations, and which results are pre-computed. Consequently, optimization can only be approximate and requires the use of techniques such as probing queries [ZL96] to collect information needed for optimization. Although estimating costs is difficult when only limited information is available, a number of cost based optimizations are described in this section. Detailed cost models have been investigated before for both XML [MW99] and relational compontents [DKS92]. Estimating costs for OLAP queries is related to the problem of estimating the size of a cube as described in [SDNR96]. The cost estimation is described in Section 10.

The optimization techniques considered here all give rise to significant performance improvements for many common types of queries. One of these techniques is based on the idea of inlining external data in predicates as was briefly introduced in Section 6.2. This is discussed more extensively in Section 9.2. We believe that the issue has not been investigated to this extent before, although the idea is briefly mentioned in [PSGJ00] for certain simple types of predicates. Also important is the problem that it may not always be possible to use the general XQuery approach to fetching decoration data from XML compontents as was presented in Section 8.3. This is true if only simpler interfaces, such as XPath, are available. Due to the flexibility of the linking mechanism used to combine OLAP and XML data, a large number of small queries may be required in such a setting. However, techniques are presented here to retrieve XML data more efficiently by combining several such queries into one, providing acceptable performance for many queries that would otherwise be unfeasible. Also discussed in this section, are the well-known techniques caching and pre-fetching of queries that has been adapted to this particular setting.

The optimization of federated queries in the SQL_{XM} system is complicated by the fact that a typical XML component used in the federation will reside on the Web, and thus it will usually have a very high degree of autonomy. A lower degree of autonomy can be assumed for the OLAP and relational compontents as these will typically be part of the same information system as the federation. On the other hand, the component types are known in advance which allows optimizations that are not possible in more general purpose federations such as Tsimmis [CGMH$^+$94]. For example, semantic heterogeneity is handled explicitly by user-defined links and hence, semantic transformation is not a concern. Also, the queries posed to XML components are always on a special form which allows special kinds of optimization.

First, we present the general cost model for federation queries, and then we explain the optimization techniques together with a refined cost model. An overview of the federated system, including these optimization techniques is provided in Section 11.

9.1 A Cost Model for Federation Queries

The cost model used in the following is based on time estimates and incorporates both I/O, CPU, and network costs. Because of the differences in data models and the degree of autonomy for the federation components, the cost is estimated differently for each component. Here, we only present the highlevel cost model which expresses the total cost of evaluating a federation query. The details of how these costs are determined for each component are described in Section 10.

As discussed earlier, the OLAP and XML components can be accessed in parallel if no XML data is used in the construction of OLAP queries. The case where XML data *is* used is discussed in the next section. The retrieval of component data is followed by computation of the final result in the temporary component. Hence, the total time for a federation query is the time for the slowest retrieval of data from the OLAP and XML components plus the time for producing the final result. This is expressed in this basic cost formula considering a single OLAP query and k XML queries:

$$Cost_{Basic} = \text{MAX}(t_{OLAP}, t_{XML,1}, \ldots, t_{XML,k}) + t_{Temp}$$

where t_{OLAP} is the total time it takes to evaluate the OLAP query, $t_{XML,i}$ is the total time it takes to evaluate the ith XML query, and t_{Temp} is the total time it takes to produce the final result from the intermediate results.

9.2 Inlining Decoration Data in OLAP Queries

As discussed in Section 6.2, references to level expressions can be inlined in predicates thereby improving performance considerably in many cases. Better performance can be achieved when selection predicates refer to decorations of dimension values at a lower level than the level to which the cube is aggregated. If e.g. a predicate refers to decorations of dimension values at the bottom level of some dimension, large amounts of data may have to be transferred to the temporary component. Inlining level expressions may also be a good idea if it results in a more selective predicate.

As Example 6.6 illustrated, level expressions can be inlined compactly into some types of predicates. Even though it is always possible to make this inlining (See Appendix B), the resulting predicate may sometimes become very long. For predicates such as "EC/EC_Link/Manufacturer/MName = Supplier/Sup_Link/SName", where two level expressions are compared, this may be the case even for a moderate number of dimension values. However, as long as predicates do not compare level expressions to measure values the predicate length will never be more than quadratic in the number of dimension values. Furthermore, this is only the case when two level expressions are compared. For all other types of predicates the length is linear in the number of dimension values. (For details, see Appendix B.) Thus, when predicates are relatively simple or the number of dimension values is small, this is indeed a practical solution. Very long predicates may degrade performance, e.g. because parsing the query will be slower. However, a more important practical problem that can prevent inlining, is the fact that almost all systems have an upper limit on the length of a query. For example, in many systems the maximum length of an SQL query is about 8000 characters. Certain techniques can reduce the length of a predicate. For instance, user defined sets of values (named sets) can be created in MDX and later used in predicates. However, the resulting predicate may still be too long for a single query and not all systems provide such facilities. A more general solution to the problem of very long predicates is to split a single predicate into several shorter predicates and evaluate these in a number of queries. We refer to these individual queries as *partial* queries, whereas the single query is called the *total* query.

Example 9.1 Consider the predicate: "EC/Manufacturer/@MCode = Manufacturer(EC)". The decoration data for the level expression is retrieved from the XML document as explained in Section 7 resulting in the following relationships between dimension values and decoration values:

EC	Manufacturer/@MCode
EC1234	M31
EC1234	M33
EC1235	M32

Using this table, the predicate can be transformed to: "(Manufacturer(EC) IN (M31, M33) AND EC='EC1234') OR (Manufacturer(EC) IN (M32) AND EC='EC1235')". This predicate may be to long to actually be posed and can then be split into: "Manufacturer(EC) IN (M31, M33) AND EC='EC1234' " and "Manufacturer(EC) IN (M32) AND EC='EC1235' ". □

Of course, in general this approach entails a large overhead because of the extra queries. However, since the query result may sometimes be reduced by orders of magnitude when inlining level expressions, being able to do so can be essential in achieving acceptable performance. Because of the typically high cost of performing extra queries, the cost model must be revised to reflect this.

The evaluation time of an OLAP query can be divided into three parts: A constant query overhead that does not depend on the particular query being evaluated, the time it takes to evaluate the query, and the time it takes to transfer data across the network, if necessary. The overhead is repeated for each query that is posed, while the transfer time can be assumed not to depend on the number of queries as the total amount of data transferred will be approximately the same whether a single query or many partial queries are posed. The query evaluation time will depend e.g. on the aggregation level and selectivity of any selections in a query. How these values are determined, is described in Section 10.

The revised cost formula for k XML queries and a single total OLAP query that is split into n partial OLAP queries is presented in the following. The cost formula distinguishes between two types of XML query results: Those that have been inlined in some predicate and those that have not been inlined in any predicate. The estimated time it takes to retrieve these results is denoted by $t_{XML,Int}$ and $t_{XML,NotInt}$, respectively. In the formula let:

- $t^{\text{MAX}}_{XML,NotInt}$ be the maximum time it takes to evaluate some XML query for which the result is not inlined in any predicate,

- $t^{\text{MAX}}_{XML,Int}$ be the maximum time it takes to evaluate some XML query for which the result is inlined in some predicate,
- $t_{OLAP,OH}$ be the constant overhead of performing OLAP queries,
- $t^{i}_{OLAP,Eval}$ be the time it takes to evaluate the ith partial query,
- $t_{OLAP,Trans}$ be the time it takes to transfer the result of the total query (or, equivalently, the combined result of all partial queries).

Then the cost of a federation query is given by:

$$Cost = \text{MAX}(t^{\text{MAX}}_{XML,NotInt}, n \cdot t_{OLAP,OH} + \sum_{i=1}^{n} t^{i}_{OLAP,Eval} + t_{OLAP,Trans} + t^{\text{MAX}}_{XML,Int}) + t_{Temp}$$

The cost formula is best explained with an example.

Example 9.2 In Figure 12, four XML queries are used, two of which are inlined in the OLAP query (XML_3 and XML_4). Hence, the OLAP query cannot be posed until the results of both these queries are returned. The inlining makes the OLAP query too long and it is split into two partial queries as discussed above. In parallel with this, the two other XML queries (XML_1 and XML_2) are processed. Thus, the final query to the temporary component, which combines the intermediate component results, cannot be issued until the slowest of the component queries has finished. In this case, the OLAP component finishes after XML_1 and XML_2, and thus, the temporary query must wait for it. □

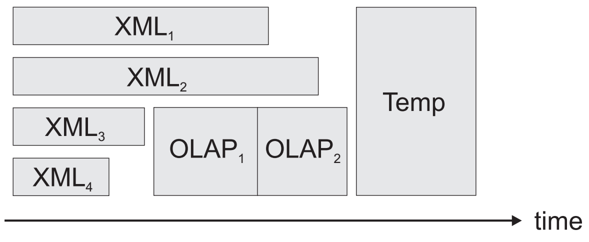
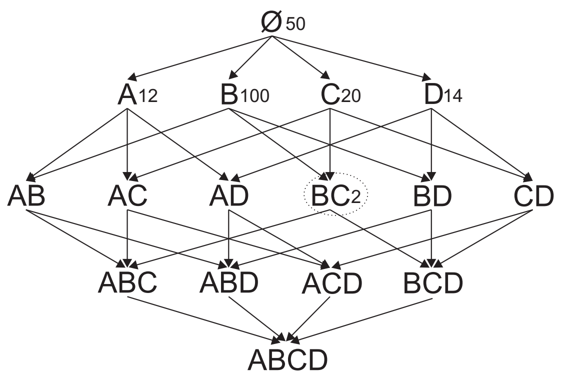


Figure 12: An example showing the total evaluation times for the component queries required to evaluate a single federation query.

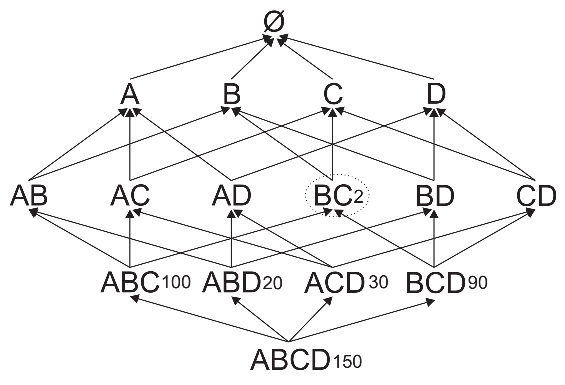
The two OLAP queries are usually not issued one after another as shown in the figure, but in parallel. Nevertheless, we made the assumption that the total evaluation time for a number of partial queries is close to the sum of the individual evaluation times. However, since the partial queries can be evaluated in parallel, the actual evaluation time will sometimes be shorter than that, e.g. due to the use of caching. The data used by one query may cause some data used by another query to be available in the cache, depending on how the data is stored in disk blocks. Working against this, is the fact that the partial queries will always work on *different* parts of data because of the way the predicates are constructed. Furthermore, the assumption is most accurate for OLAP systems running on machines with only a single CPU and disk. For machines with multiple CPUs and disks the maximum evaluation time for any partial query may provide a better estimate. Often the best general estimate will be somewhere in between, and thus, an average value can be used. Also, the evaluation time for the total query will sometimes provide a good estimate. It will always take longer than any partial query, because the partial queries are more selective, and it will generally be faster than the sum of the partial evaluation times, because optimization can be more effective when a single query is used. For example, a full table scan may be the fastest way to find the answer to the total query, but by posing several partial queries a number of index lookups may be used instead. Using the sum of the partial evaluation times will generally be sufficiently accurate, because, as is further discussed in Section 10, we cannot assume to have detailed cost information available, and consequently, estimates can only be approximate. Which of these estimates is best for a particular OLAP system can be specified as a tuning parameter to the federation.

Since any subset of the level expressions can be inlined in the OLAP query, the number of inlining strategies is exponential in the number of level expressions. None of these can be disregarded simply by looking at the type of predicate and estimated amount of XML data. Even a large number of OLAP queries each retrieving a small amount of data may be faster than a few queries retrieving most or all of the stored data. Further complicating the issue, is the fact that the choice of whether a particular level expression should be inlined may depend on which other expressions are inlined. Consider e.g. two predicates that both refer to decorations of values at a low level in the cube, and hence, require the retrieval of a large part of the cube. Inlining only one of them may give only a small reduction in the OLAP result size, because the low level values must still be present in the result to allow the other decoration to be performed. For the same reason, we cannot consider XML data used for selection independently from XML data that are only used for decoration or grouping. Also, a level expression that occurs multiple times in a predicate need not be inlined for all occurrences.

When adding a level expression to the set of inlined expressions, the total cost may increase or it may decrease. An increase in cost can be caused by two things: The OLAP query may have to wait longer for the extra XML data, or more OLAP queries may be needed to hold the extra data. Any decrease in cost is caused by a reduction in the size of the OLAP result, either because the selectivity of the predicate is reduced or because a higher level of aggregation is possible. A smaller OLAP result may reduce both the OLAP evaluation time and the temporary evaluation time.



(a)



(b)

Figure 13: Top-down (a) and bottom-up (b) generation of inlining strategies for the four level expressions A, B, C, and D. The numbers represent the cost of a strategy, and the optimal solution is indicated by a dashed circle.

Even though there is an exponential number of inlining strategies, this will almost never be a problem, as selections typically contain only a few level expressions. Thus, performing an exhaustive search is an adequate solution to this problem. A few heuristics are used to reduce the search space even more: If a combination of level expressions produces a cost that is much larger than the combinations previously generated, it is not used to generate further combinations. For this to be a valid heuristic, it is important to begin from the full set of level expressions and remove each one iteratively until none are left. Using the opposite approach, i.e. beginning from the empty set and add elements iteratively, the heuristic will sometimes fail. Consider e.g. the two approaches shown in Figure 13, where each letter represents a level expression and the values shown for some of the combinations represent costs. A top-down approach, beginning from the empty set of expressions, is shown in Figure 13(a) and a bottom-up approach, beginning from the full set, in Figure 13(b). A heuristic top-down approach ignores all combinations containing a certain level expression if it is very expensive to inline. For instance, all combinations containing B would be ignored in Figure 13(a). However, even though an expression is much more expensive to inline than the other expressions, this may be compensated for by a much higher level of aggregation in the OLAP query. This may only be achieved when the expensive level expression is inlined together with other expressions as discussed above. Hence, the optimal cost, represented by the dashed circle, may actually contain expression B, and consequently, the top-down approach will fail to find the optimal value. A bottom-up approach is less likely to fail because it first considers the strategy where all expressions are inlined. Thus, if any combinations give a special reduction in the cost, this is taken into account when selecting which combinations to disregard.

Certain situations may still cause the bottom-up approach to fail, but a refinement to the heuristic can often prevent this. Consider the situation in Figure 13(b). Here, the optimal strategy may be skipped because all the combinations leading to it are very expensive. This can only occur if all the level expressions not in the optimal strategy take very long time to evaluate or produce a high number of OLAP queries. In Figure 13(b), both A and D would have to be very expensive. First, this makes it likely that the other combinations, which also involve A or D, are expensive too. Second, the heuristic can be refined to handle this problem by identifying when a single level expression causes a long waiting time or many OLAP queries.

The optimization approach can be summarized as follows: Generate all inlining strategies bottom-up except for combinations with a very high cost. Thus, if the cost is high for a particular combination, no subsets of the combination are considered. However, if the high cost is mainly caused by a single level expression, this restriction does not apply and the subsets of the combination are considered anyway. What constitutes a "very high cost", is determined dynamically based on the number of level expressions. Thus, for only few expressions almost all combinations are considered, whereas for more expressions the heuristic is used more aggresively. This can be generalized to cope with a higher number of level expressions by choosing only a fixed number of combinations at each level. This reduces the time complexity from exponential to quadratic in the number of level expressions.

Although predicates in a query will typically contain only a few level expressions, a higher number is, of course, possible. If the number of level expressions n is too high, then the optimal solution cannot be found within reasonable time. In fact, the problem of finding the minimal cost is NP-hard as will be shown in the following. It is well known that finding the global minimum of a complex cost function in a large search space with a high number of local minima is NP-hard [SSV98]. In the following, we will use $Cost(S)$, where S is a set of level expressions, to mean the value of the $Cost$ function when inlining the expressions in S.

Theorem 9.1 Finding the global minimum of the cost function $Cost$ is NP-hard.
Proof outline: (Proof by example)
Let S be a set of level expressions. $Cost(S)$ is a local minimum if for all level expressions $x_i \notin S$ and $x_j \in S$ the following is satisfied: $Cost(S) < Cost(S \cup \{x_i\})$ and $Cost(S) < Cost(S \setminus \{x_j\})$. Such situations *can* occur. Consider e.g. this example:

- Inlining each x_i increases the number of OLAP queries without an equivalent reduction in the size of the OLAP result.
- All x_j in S decorate the same level and thus, inlining only a subset of them prevents the cube from being aggregated to a higher level.

Moreover, these local minima can exist independently of each other, e.g if there is no overlap between their elements. Hence, a large number of minima can exist if the search space is large. □

A well known and relatively simple technique to find a good solution to this sort of problem is *Simulated Annealing* [KGV83, IW87]. A Simulated Annealing algorithm attempts to find the global minimum by considering only part of the search space, and behaves much like a greedy algorithm. However, with a certain probability transitions to a state with a higher cost than the previous one are allowed, which enables the algorithm to escape a local minimum. The propability with which these "up-hill moves" are allowed, decreases over time. As stated earlier, the search space will typically be small and hence, the application of techniques such as Simulated Annealing to this specific optimization problem is outside the scope of this paper.

9.3 Optimizing XML Data Retrieval Through Limited Query Interfaces

As explained in Section 8.3 a single XQuery query can be used to fetch all decoration data from each XML component. However, such a highlevel query language cannot generally be assumed when data is accessed over the Web, e.g. through a URI. Also, systems such as the Tamino XML Database currently only provides an XPath-like interface [AG01b]. The XPath language is sufficient for applications such as in the SQL_{XM} syntax, where it provides a compact and simple way to identify nodes in an XML document, but the language exhibits great limitations when used for querying data sources [BC00]. In this section, we investigate methods for retrieving decoration data efficiently when only an XPath interface is available. Although we focus on XPath, the optimization methods presented here can easily be applied to other comparable languages like e.g. XQL [RLS98]. A related problem is investigated in [GMY99] which considers queries through limited interfaces. These interfaces may e.g. require certain data values to be specified in the query in order to retrieve the result.

Having only an XPath interface, decoration data can be fetched by posing a query for each dimension value as described in Section 8.3. If the XML component is specifically optimized for this and the primary performance bottleneck is the transfer of result data, this method may provide acceptable performance. However, in general the overhead of transferring, parsing and evaluating a large number of queries will be too expensive. A general technique to reduce this cost is to combine groups of queries into a single query. For instance, a number of queries each containing a predicate can sometimes be combined to a single query containing the disjunction of the predicates. The result of this combined query will then have to be split up locally. Consider e.g. the following example:

Example 9.3 Recall the definition of EC_link in Example 4.2. The XML data for the level expression "EC[ALL]/EC_link/Description" can be retrieved by issuing these three XPath expressions:

/Components/Supplier/Class/Component[@CompCode = 'EC1234']/Description
/Components/Supplier/Class/Component[@CompCode = 'EC1235']/Description
/Components/Supplier/Class/Component[@CompCode = 'EC2345']/Description

These expressions can be combined into a single expression retrieving the Description nodes: /Components/Supplier/Class/Component[@CompCode = 'EC1234' OR @CompCode = 'EC1235' OR @CompCode = 'EC2345']/Description. □

The problem with this approach is that the resulting nodes cannot always be distinguished in the result. For this to be possible, both the locator and the user defined XPath expression must be present. This is necessary because according to the XPath Recommendation [W3C99] the result of an XPath expression is an *unordered* collection of nodes, and hence, there is no way to identify each decoration node, except by using the locator. Furthermore, an XPath expression cannot change the structure of a document, but only return a set of existing nodes from it. Consequently, the result of an expression can only contain *entire* nodes and not partial nodes. Thus, to maintain the relationship between the locator and the user defined XPath expression, their common parent node must be retrieved in its entirety.

Example 9.4 Considering the Components document, a single XPath expression cannot fetch only the CompCode and Description children providing a result such as:

```
<Component CompCode="EC1235">
 <Description>16-bit flip-flop</Description>
</Component>
```

It *is* possible to fetch only the CompCode and Description nodes using an expression such as /Components/Supplier/Class/Component/@CompCode | /Components/Supplier/Class/Component/Description. However, the result contains only an unordered set of CompCode and Description nodes that cannot be used to determine which pairs of nodes belong together. Thus, the entire Component nodes including all children must be fetched instead of just the Description nodes using the expression /Components/Supplier/Class/Component[@CompCode = 'EC1234' OR @CompCode = 'EC1235' OR @CompCode = 'EC2345']. The Description nodes can then be matched with their corresponding CompCode nodes by applying three expressions locally, each similar to this: /Component[@CompCode = 'EC1234']/Description. This means that instead of using three expressions to retrieve the Description nodes, a single expression is used to retrieve the Component nodes. Notice that if the CompCode attribute were part of the Description node or if the entire Component nodes were needed, no additional data would be fetched. □

This example illustrates that combining multiple expressions may result in the retrieval of extra data. Still, this approach may be much faster than evaluating each expression individually. In fact, it may be advantageous to combine multiple expressions even if this means retrieving much more data. For example, a large part or even the whole document can be fetched and queried locally. The choice between evaluating many small expressions and combining the expressions, possibly retrieving too much data, is decided by comparing the estimated cost of each strategy. Details of how to estimate the cost of an XPath expression is presented in Section 10.2.

The strategy of combining XPath expressions is valid for both enumerated and natural links. However, because XPath expressions constructed from a natural link is always on the special form discussed in Section 4 and in the example above, creating a combined expression is straightforward compared to the general case. For enumerated links the high degree of flexibility means that each dimension value may be associated with a unique XPath expression. In the worst case each expression refers to a unique XML document. However, this is unlikely to occur in practice, and it will usually be possible to combine most of the expressions.

Example 9.5 Assume that the manufacturer codes were not present in the Components document. Then the enumerated link Manuf_link could be defined as: $\{($"M31", "components.xml", "/Components/Supplier[@SCode='SU13']/Class/Component[@CompCode='EC1234']/Manufacturer"$)$, $($"M32","components.xml", "/Components/Supplier/Class/Component[@CompCode='EC1235']/Manufacturer"$)\}$. In order to retrieve the decoration data for the level expression Manufacturer[ALL]/Manuf_link/MName these two XPath expressions could be used individually: /Components/Supplier[@SCode='SU13']/Class/Component[@CompCode='EC1234']/Manufacturer/MName and /Components/Supplier/Class/Component[@CompCode='EC1235']/Manufacturer/MName.

The two expressions cannot simply be combined by creating a disjunctive predicate. In this case, the entire Supplier nodes must be retrieved, because the second XPath expression does not select any particular Supplier nodes. □

The following algorithm combines a set of XPath expressions XP to a single expression that may retrieve additional data compared to evaluating the individual expressions. For simplicity the expressions are assumed to be on the form $/node_1[predicate_1]/node_2[predicate_2]/\ldots/node_k[predicate_k]$, that is, without any wildcards, shorthand notations or union expressions, and allowing only downwards movement in the XML tree. Recall that, a (location) *step* of an XPath expression is the element name selected between each pair of "/".

Algorithm 9.3.1 Combine XPath Expressions

```
1 CombineXPathExpressions(XP, StepNo)
2   Step := GetNextStep(xp$_1$, StepNo)
3   for all xp$_i$ $\in$ XP:
4     Step$_i$ := GetNextStep(xp$_i$, StepNo)
5     if Step$_i$ $\neq$ Step or Step$_i$ contains a predicate then Done := true
6   if Done then return CommonPredicate(XP, StepNo)
7   else return Concat(Step, CombineXPathExpressions(XP, StepNo+1))
```

Algorithm 9.3.1 combines all XPath expressions in XP into a single expression. The function CommonPredicate combines the predicates at the same step of a number of XPath expressions, e.g. by creating the disjunction of the predicates.

Given a set of XPath expressions, any subset of these can be combined resulting in more than one expression. Hence, there is an exponential number of ways to combine the expressions, one of which may be faster than the other. Providing a general algorithm for finding an optimal or good solution to this problem is outside the scope of this paper. Instead, we simply estimate the cost for three situations: When evaluating the expressions *individually*, when combining *all expressions* referring to the same document, and when combining all expressions with predicates at the *same location step*. The latter is done using a slightly modified version of Algorithm 9.3.1. Although not an optimal solution, this avoids evaluating many expressions if the overhead of doing so is very high, and also avoids retrieving too much data if the data transfer rate is low.

To summarize, three different strategies are used when evaluating a set of XPath expressions resulting from a level expression: combining none, some, or all of the expressions using Algorithm 9.3.1. If the level expression is based on a natural link, it is always possible to combine all the expressions which typically produces a low overhead expression because of the way these links are defined. If it is based on an enumerated link, combining all expressions to a single expression may retrieve all or a large part of the document. Hence, we also consider the situation where only expressions having predicates at the same location step are combined. For each of these three strategies, the total evaluation cost is estimated and the cheapest one is used.

9.4 Caching and Pre-fetching

Another technique that can significantly reduce the total query evaluation time is caching. Two types of results should be considered for caching: Results of complete federation queries as produced by the temporary component and intermediate results produced by the OLAP and XML components. In this section we discuss how and when to use cached data for these types of results. Because of the potentially large performance gains that can be achieved by storing results locally, it will often be a good idea to pre-fetch certain data that are likely to be needed. The main question regarding pre-fetching is what data to fetch, which is discussed briefly. We will begin by looking at caching of intermediate results.

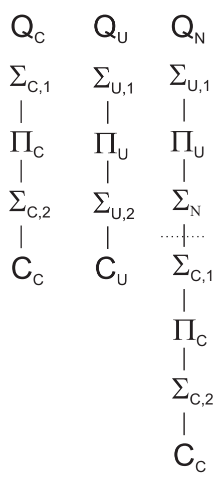
Basically, caching results of OLAP queries is done by keeping the otherwise temporary tables that are created in the temporary component. Associated with each such table is the part of the query tree that produced the table. Given a new query tree, it is determined whether the cached result is identical to or can be used to produce the new result. When this is the case, the cost of using the cached result is compared to the cost of not using it. If the query that produced the cached result is identical to the new query, it will always be cheaper to use the cache. However, if e.g. extensive aggregation is needed on the cached result to produce the final result, it may be cheaper to fetch a possibly pre-aggregated result from the OLAP component.

Determining whether a cached result can be used to produce a new result must be done efficiently since a large cache may hold many different results. This is done by performing a few simple tests on the query tree corresponding to each cached result. Consider the two query trees Q_C and Q_U in Figure 14(a), representing the *cached* and the *user* query, respectively. If the cached result can be used to produce the needed result then Q_U must be expressible in terms of Q_C. The tests discussed next will determine whether this is possible, but let us first see how the final query is constructed in the non-trivial case where Q_U and Q_C are not identical. (By identical we mean that they contain the same operations, disregarding the order of selections in each Σ block.)

Q_C must form the bottom part of the *new* query Q_N that is constructed from Q_U, and the part of Q_U that restricts the result further must be applied to this bottom part. The resulting query Q_N is also shown in Figure 14(a). Two observations can be made here. First, the selections in $\Sigma_{U,2}$ are divided into two groups: Those that can be pushed below Π_C and those that cannot. The first group must contain all selections in $\Sigma_{C,2}$, while the latter must contain all the selections in $\Sigma_{C,1}$. Any remaining selections are placed above $\Sigma_{C,1}$ in Σ_N. Second, we see that $\Sigma_{U,1}$ is preserved at the top. This is always the case because only selections that cannot be pushed below the GP are left above it as described in Section 8.2.

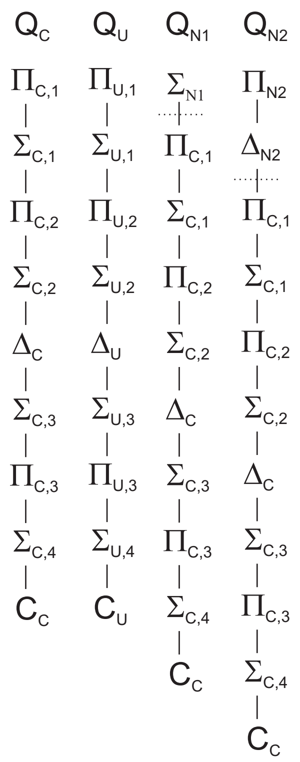
The transformation of Q_U to Q_N is possible if the following three requirements are all satisfied:

1. $C_C = C_U$. The component cubes must be the same.



Q_C Q_U Q_N
$\Sigma_{C,1}$ $\Sigma_{U,1}$ $\Sigma_{U,1}$
Π_C Π_U Π_U
$\Sigma_{C,2}$ $\Sigma_{U,2}$ Σ_N
C_C C_U $\Sigma_{C,1}$
Π_C
$\Sigma_{C,2}$
C_C

(a)



Q_C Q_U Q_{N1} Q_{N2}
$\Pi_{C,1}$ $\Pi_{U,1}$ Σ_{N1} Π_{N2}
$\Sigma_{C,1}$ $\Sigma_{U,1}$ $\Pi_{C,1}$ Δ_{N2}
$\Pi_{C,2}$ $\Pi_{U,2}$ $\Sigma_{C,1}$ $\Pi_{C,1}$
$\Sigma_{C,2}$ $\Sigma_{U,2}$ $\Pi_{C,2}$ $\Sigma_{C,1}$
Δ_C Δ_U $\Sigma_{C,2}$ $\Pi_{C,2}$
$\Sigma_{C,3}$ $\Sigma_{U,3}$ Δ_C $\Sigma_{C,2}$
$\Pi_{C,3}$ $\Pi_{U,3}$ $\Sigma_{C,3}$ Δ_C
$\Sigma_{C,4}$ $\Sigma_{U,4}$ $\Pi_{C,3}$ $\Sigma_{C,3}$
C_C C_U $\Sigma_{C,4}$ $\Pi_{C,3}$
C_C $\Sigma_{C,4}$
C_C

(b)

Figure 14: Determining when a cached result can be used for OLAP queries and federation queries.

2. Let $\Pi_C = \Pi_{[L_C]<F(M_C)>}$ and $\Pi_U = \Pi_{[L_U]<F(M_U)>}$. Then $L_C \subseteq L_U \wedge M_C \subseteq M_U$ must be satisfied. This is necessary because it must be possible to roll up from Q_C to Q_U. Furthermore, the roll-up from the levels in C_C to L_C must be strict, since it should be legal to roll further up to L_U.
3. $\Sigma_{C,2} \subseteq \Sigma_{U,2}$ and all $\sigma \in \Sigma_{U,2} \backslash \Sigma_{C,2}$ can be pulled above Π_C. The remaining selections must contain $\Sigma_{C,1}$, i.e. $\Sigma_{C,1} \subseteq \Sigma_{U,2} \backslash \Sigma_{C,2}$. This leaves $\Sigma_N = (\Sigma_{U,2} \backslash \Sigma_{C,2}) \backslash \Sigma_{C,1}$, which is performed in the temporary component.

All these requirements can be tested efficiently because no transformations need to be done on either of the trees. Only simple comparisons of the operations are performed and the number of comparisons is quadratic in the number of selections, typically for only a few selections.

Intermediate XML results can also be cached, either by storing the temporary tables relating dimension values and decoration values, or by storing the XML nodes that are retrieved. The benefit of storing the unprocessed XML data is that too much data may have been retrieved as described in Section 9.3 and this extra data may be requested at a later time. However, the decoration data must be processed and stored in a table each time it is used, which is avoided if the decoration tables are retained. This local processing can be expected to be much faster than retrieving the decoration data from its source and the performance gain will still be significant by storing the raw XML data. Hence, this approach is used if temporary data storage is scarce, otherwise, both approaches are used. For each cached XML result the corresponding query is stored. By comparing this query with the user query, it can be determined whether the cached result can be used. This comparison is very similar to the one performed in Algorithm 9.3.1.

If only limited storage is available for caching, it may be necessary to choose between caching of OLAP data and caching of XML data. Which of these choices provides the best performance depends e.g. on the amount of data, the communication delay, and whether or not the temporary component is located on the same server as the OLAP component. Thus, no general rule can be made about which is better, but this must be decided for each component e.g. by the DBA.

Caching of entire federation queries can potentially provide a larger performance gain in case of a cache hit than caching of intermediate queries. However, an intermediate result can be used in different federation queries and is more likely to be useful than that of an entire query. Also, it is more expensive to determine whether a cached result of an entire query can be used to produce the result of another query. As for OLAP queries, the optimal solution to this problem is to decide whether the incoming query can be transformed such that the cached query constitutes the bottom part of it. However, there are no simple list of requirements that can be tested to determine exactly when this is possible. Simple requirements can be listed to identify some special situations where it is possible, though, and still achieve significantly better than requiring the two trees to be identical. Because of the time overhead of comparing entire queries, and because computing the final result from the intermediate component results will typically not be a performance bottleneck, we do not cache entire federation queries. Hence, it is out of this paper's scope to make a complete analysis of these requirements. Instead, we present a few heuristics to illustrate the idea.

Consider the two queries Q_C and Q_U in Figure 14(b). One way to loosen the requirement that the trees should be identical, is to say that Q_U is allowed to contain selections that are not in Q_C, but only if they can be pulled above $\Pi_{C,1}$. The result of this is query Q_{N1} shown in Figure 14(b). If this is possible the selection only refers to levels in $\Pi_{C,1}$ and hence, it can also be pulled over all GPs below $\Pi_{C,1}$, as well as any decorations. Since a selection can always be pulled above other selections, it is sufficient to compare the extra selections to $\Pi_{C,1}$. Extra decorations in Q_U can also be pulled above $\Pi_{C,1}$ if they are not ALL decorations and are not used in selections. This situation is illustrated by query Q_{N2} in Figure 14(b).

All cached results expire after a certain time. This time depends on the specific type of data and can be specified as a tuning parameter by the DBA or, for XML components, by the creator of a link. When the cache fills up, a *least recently used* replacement strategy is used.

Because retrieving data over a network will often be a bottleneck in the federation, pre-fetching relevant component data and storing it locally can improve performance considerably. The central question is what to fetch. Entire federation queries can be pre-fetched, but unless the same or very similar queries are asked repeatedly, keeping a store of results is not likely to provide many hits. Also, the same difficulties as described above for caching apply here. Consequently, we do not pre-fetch entire federation queries. Pre-fetching component data is more likely to result in a high hit-rate, because the same data can be used in many different federation queries. OLAP data can be pre-fetched in different roll-up combinations and used like cached results. Statistics, about which combinations of levels are most often used in queries, are used to select which combinations should be pre-fetched. How to obtain these statistics is discussed further in Section 10.1. For XML components, the entire subtrees pointed to by links can be retrieved in advance on a periodical basis. This means that all decoration data will be available locally and may improve performance significantly if access to XML components is slow. If only limited temporary storage is available, decoration data is retrieved for only the most frequently used links. A more sophisticated strategy could also take the access times of XML components into account.

It may also be a good idea to retrieve additional data during the evaluation of a federation query if this improves the likelihood that the data can be reused later. Hence, if the selectivity for an OLAP or XML query is estimated to be more than 20%, we retrieve the data without performing selection. This may reduce the future response times significantly at only a small extra cost. The estimation of selectivity is discussed in Section 10.1.

In summary, we perform caching and pre-fetching for component queries only. Intermediate OLAP results stored in temporary tables as well as raw XML data are kept for a certain amount of time, which can specified as a tuning parameter. If adequate storage is available, temporary XML tables are also stored to avoid constructing the same tables again. Currently, we do not cache or pre-fetch entire federation queries. A *least recently used* replacement strategy is used. As will be described in Section 12, experiments indicate that large cost reductions can be achieved using these techniques.

10 Determining Cost Parameters

This section describes the estimation of the cost parameters that were used in the cost formulas in Section 9.2. This cost information is collected by the *Statistics Manager* and used by the cost evaluators in Figure 9.

The amount of cost information available to the federation may vary between federation components. Du et al. [DKS92] distinguish between three types of federation components: *Proprietary*, where all cost information is available to the federation, *Conforming*, where the component DBMS provides the basic cost statistics, but not full information about the cost functions used, and finally, *Non-Conforming*, where no cost information is available. Usually, only the DBMS vendor has full access to all cost information, and hence, we consider only conforming and non-conforming components here. A primary motivation for the federation of OLAP and XML components is to provide easy access from e.g. a corporate OLAP database to XML data available from restricted sources such as the Internet. Thus, a high degree of autonomy must be expected for XML components, while OLAP components may or may not provide access to cost information. Because of this, we assume in the following that the OLAP component is either conforming or non-conforming, while the XML components are non-conforming. The temporary component used by the federation is assumed to be a conforming component.

Several techniques have been used in the past for acquiring cost information from federation components. A commonly used technique is *query probing* [ZL96] where special queries, called *probing queries*, are used to determine cost parameters. For proprietary or conforming components, cost data, such as selectivities and data cardinalities, can simply be fetched from the available meta data. If this information is not available, probing queries can instead retrieve samples of the component data. Typical information that can be determined in this way includes maximum and minimum values, cardinalities, and data element sizes as well as the time required to evaluate the probing query. In general the latter type of queries will provide less accurate information than when it is gathered from the component's own statistical data. *Adaptive cost estimation* [LTD95] is used to enhance the quality of cost information based on the actual evaulation costs of user queries.

Because different requirements and difficulties are exhibited by the three types of components (OLAP, XML, and relational) different techniques are used to determine their cost information. Hence, the presentation of cost parameters is divided into three sections corresponding to each of the three component types.

10.1 Determining OLAP Cost Parameters

As described earlier the cost of an OLAP query comprises a constant query overhead that does not depend on the particular query being evaluated, the time it takes to actually evaluate the query, and the time it takes to transfer data across the network if necessary:

$$t_{OLAP} = t_{OLAP,OH} + t_{OLAP,Eval} + t_{OLAP,Trans}$$

The statistical information that is needed to estimate these parameters, may be available from the OLAP component's meta data, in which case it is used directly. However, if such data is not available, we use probing queries to determine the statistical information and continuously update it by measuring the actual cost of all queries posed to the components. The probing queries are all relatively inexpensive and can be posed when the system load is low, and the overhead of adapting the cost information to the actual costs is insignificant. Hence, these methods introduce little overhead on the federated system.

In the following, we explain how the cost parameters are estimated given an OLAP query by using probing queries, and how to adapt the cost information when the actual cost of a query is found. We do not explicitly consider the use of existing statistical information as this depends very much on the specific DBMS and is similar to the use of information determined using probing. More specifically, we describe the statistical information that the estimation is based on and how it is obtained, how the three cost parameters are estimated using this information, and how the information is updated.

10.1.1 Statistical Information

The estimation of cost parameters is based on statistical information represented by the functions shown in Table 3.

Function	**Description**
$NetworkDataRate(C)$	The rate with which data can be transferred from cube C to the temporary component
$DiskDataRate(C)$	The rate with which data can be read from disk for cube C
$Selectivity(\theta, C)$	The selectivity of predicate θ evaluated on cube C
$FactSize(M)$	The size of measures M
$RollUpFraction(\mathcal{L}, C)$	The relative reduction in size when rolling cube C up to levels $\mathcal{L}$
$Size(C)$	The size of cube C
$EvalTime(Q)$	The time for evaluating query Q

Table 3: Statistical functions used to determine OLAP cost parameters.

These functions are explained in the following:

$NetworkDataRate(C)$**:** This function returns the rate with which data can be transferred from cube C to the temporary component if this is not located on the same server as the OLAP component. This is estimated by posing a probing query to C. The result is measured in size and timed from when the first result tuple is received until the last result tuple is received. The data rate can then be approximated from the measured size and time.

$DiskDataRate(C)$**:** This function returns the rate with which data can be read from disk for cube C. This can be estimated by posing a probing query that retrieves a part of the base cube and measuring the size of the result as well as the total query evaluation time: $DiskDataRate(C) = \frac{Size(Q_{Probe}(C))}{t_{Q_{Probe}} - t_{OLAP,OH} - t_{OLAP,Trans,Q_{Probe}}}$. By subtracting the constant query overhead and the estimated transfer cost, which will be described later, only the evaluation time is left. The assumption is that a large part of the evaluation time for a query that retrieves data from the base cube is spent reading the result data from disk. Because indexes are typically used to locate such data, little additional data is read from disk.

$Selectivity(\theta, C)$**:** This function returns the fraction of the total size of C that is selected by θ. This is estimated using standard methods, i.e. by assuming a uniform distribution, and by considering cardinality, minimum and maximum values of the involved attributes[EN00]. E.g., if $\theta =$ "$Level \leq k''$, where k is a constant, the selectivity of θ can be estimated by $Selectivity(\theta, C) = \frac{k - min(Level)}{max(level) - min(Level)}$. (Often the smallest/largest but one is used to ignore extreme values.) If information about cardinality, minimum and maximum values is not available, it is obtained by posing probing queries that explicitly request this information.

$FactSize(M)$**:** This function returns the size in bytes of a fact containing only values for the measures in M. This is based on the average size of a measure value which is determined from a single probing query. $FactSize(C)$ is used to refer to the size of a fact containing all measures in C.

$RollUpFraction(\mathcal{L}, C)$**:** This function returns the fraction to which C is reduced in size, when it is rolled up to the levels $\mathcal{L}$. Shukla et al. [SDNR96] propose three different techniques to estimate the size of multidimensional aggregates without actually computing the aggregates: One is based on the assumption that facts are distributed uniformly in the cube and does not consider the actual contents of the cube, one performs the aggregation on samples of the cube data and extrapolates to the full cube size, and one scans the entire cube to produce a more precise estimate. Here we use the first method because of its simplicity and speed, and because experimental results in [SDNR96] show that it performs well even when facts are distributed rather non-uniformly.

Using this method, the size of an aggregated result is given by the following standard formula for computing the number of distinct elements d obtained when drawing r elements from a multiset of n elements: $d = n - n(1 - \frac{1}{n})^r$. Given a GP $\Pi_{[\mathcal{L}],<F(M)>}(C)$ we can then estimate the size of the result letting $n = \|L_1 \times L_2 \times \cdots \times L_k\|$ for all $L_i \in \mathcal{L}$ and r be the number of facts in C, i.e. $r = \frac{Size(C)}{FactSize(C)}$. Hence, $RollUpFraction(\mathcal{L}, C) = \frac{d}{r}$.

$Size(C)$**:** This function returns an estimated size of C, where C may be a cube resulting from an OLAP query, denoted as $C = Q(C')$. The size of $Q(C')$ depends on the selectivity of the predicates included in the query, and to which levels the cube is rolled up. This leads to the following:

$$Size(Q(C')) = \begin{cases} Selectivity(\theta, C') \cdot Size(Q'(C')) & \text{if } Q = \sigma_\theta(Q') \\ RollUpFraction(\mathcal{L}, C') \cdot \frac{FactSize(M)}{FactSize(C)} \cdot Size(Q'(C')) & \text{if } Q = \Pi_{[\mathcal{L}],<F(M)>}(Q') \\ \text{Size of fact table} & \text{if } Q(C') \text{ is the base cube.} \end{cases}$$

$EvalTime(Q)$**:** This function returns the estimated time it takes to evaluate the query Q in the OLAP component. OLAP components often use pre-aggregation to enable fast response times. Hence, the evaluation of a query can be divided into three strategies, the choice of which is determined by which aggregations are available in the cube. The cost of each of these strategies are discussed in the following. We first present the general formulas that may be used to calculate the evaluation time of an OLAP query, and then we describe how they are used and when they are applicable.

First, a pre-aggregated result may be available that rolls up to the same levels as the query being evaluated. In that case, the evaluation of the query reduces to a simple lookup. Assuming that proper indexes are available on dimension values, any selections referring to dimension values in the aggregated result can be evaluated using these indexes. This means that the evaluation time of such a query can be assumed to be directly proportional to the combined selectivity of selections in the query and to the size of the pre-aggregated result. However, such a pre-aggregated result can only be used if it is available and if no selection refers to measures in the unaggregated cube or to levels that has been aggregated away in the pre-aggregated result.

If these requirements are not satisfied, the cube may follow a second strategy, in which no pre-aggregated results are used. Instead, the query is computed entirely from the base cube. The cost of this computation depends, of course, on the algorithms implemented in the DBMS, but a simplified cost formula is used, that reflects the evaluation capabilities of many OLAP databases. As above, we assume that selections can be evaluated efficiently by use of proper indexing. Hence, any selections in the query that refers to levels in the cube can be evaluated while accessing the cube, such that only facts that satisfy the selection predicates are read from disk. Let this data amount be denoted by d. Any selections that refer to measures in the unaggregated cube can be evaluated at the same time, but these cannot be assumed to make use of indexes. Let the resulting amount of data be denoted by d'. A widely used method of performing aggregation is hashing, see e.g. [Gra93, GBC98], and we assume that a simple hashing strategy is used. Hence, d' bytes of data is partitioned using a hash function and written to disk. Each partition is then read, while aggregation as well as any selections referring to the aggregated values are performed on the fly. Hence, $d + 2d'$ bytes are read from disk to produce the final result.

A third strategy can be used when no selections refer to measures in the unaggregated cube, but one or more selections may refer to levels that are not present in the aggregated result. In that case the first strategy cannot be used. However, any pre-aggregated result can still be used as long as it allows all selections that do not refer to measures in the aggregated result to be evaluated. Further aggregation must be performed as described for the second strategy to produce the final aggregated result. Hence, the same cost formula can be used, but now the initial cube is instead pre-aggregated and no selections refer to measures. Hence, $d' = d$ and $3d$ bytes are read.

These strategies are summarized in the following formula for the evaluation time of an OLAP query. Assume that queries are on the form $Q(C) = \Sigma_2(\Pi_{[\mathcal{L}]<F(M)>}(\Sigma_1(C)))$, where Σ_i denotes a sequence on selections. Let $S_{\mathcal{L}} = \prod_{j=1}^{k} Selectivity(\theta_j, C)$, where selection predicates $1, \ldots, k$ refer to levels in C, and let $S_M = \prod_{j=1}^{l} Selectivity(\theta_j, C)$, where selection predicates $1, \ldots, l$ appear in Σ_1 and refer to measures in C. We use the abbreviation DDR for $DiskDataRate$.

$$EvalTime(Q(C)) = \begin{cases} DDR(C)^{-1} \cdot d \quad \text{where } d = S_{\mathcal{L}} \cdot Size(\Pi_{[\mathcal{L}]<F(M)>}(C)) \\ \quad \text{if no selections in } \Sigma_1 \text{ refer to measures or levels not in } \Pi_{[\mathcal{L}]<F(M)>}(C), \\ \quad \text{and } \Pi_{[\mathcal{L}]<F(M)>} \text{ is pre-aggregated} \\ DDR(C)^{-1} \cdot (d + 2d') \quad \text{where } d = S_{\mathcal{L}} \cdot Size(C) \text{ and } d' = S_M \cdot d \\ \quad \text{if any selections in } \Sigma_1 \text{ refer to measures} \\ DDR(C)^{-1} \cdot 3d \quad \text{where } d = S_{\mathcal{L}} \cdot Size(\Pi_{[\mathcal{L}']<F(M)>}(C)) \\ \quad \text{for } \Pi_{[\mathcal{L}]<F(M)>}(C) \rightarrow \Pi_{[\mathcal{L}]<F(M)>}(\Pi_{[\mathcal{L}']<F(M')>}(Q(C))) \\ \quad \text{if no selections in } \Sigma_1 \text{ refer to measures, or to levels} \\ \quad \text{not in } \Pi_{[\mathcal{L}']<F(M')>}(C), \text{ and } \Pi_{[\mathcal{L}']<F(M')>} \text{ is pre-aggregated} \end{cases}$$

The problem with this formula is that, if a non-conforming component is used, it is not possible to know which results are pre-aggregated and which are not. Hence, we cannot always distinguish between fully and partially pre-aggregated results, and for partially pre-aggregated results we cannot know *which* results are used. To handle this problem, we use an adaptive approach based on the actual costs of performing the OLAP query. Still, an initial guess is made at the level of pre-aggregation used to evaluate a query. This could be based on the pessimistic assumption that no pre-aggregated results are present in the cube and thus, get a cost that is likely to be too high. Alternatively, it could be based on the optimistic assumption that the optimal pre-aggregated result is available and get a cost that is likely to be too low. However, it is not possible to say which of these are best, even for a specific purpose such as deciding whether or not to inline XML data as discussed earlier. The reason for this is, that no simple relationship exists between $EvalTime$ and the amount of inlining performed. Whether a low estimate for $EvalTime$ will cause more or less XML data to be inlined depends e.g. on the size and selectivities of predicates and the cost of performing computation in the temporary component. Hence, no simple guidelines can be given as to whether it is better to use an optimistic or a pessimistic estimate of $EvalTime$. Instead, a level of pre-aggregation is chosen between the bottom and top level of each dimension and this choice is then improved by moving up or down in the dimensions as the actual cost of the query is measured. If the actual cost is larger than the estimate, too high a level of pre-aggregation is assumed and vice versa. This adaptive technique is explained in more detail later.

10.1.2 Estimating Cost Parameters

The cost parameters $t_{OLAP,OH}$, $t_{OLAP,Eval}$, and $t_{OLAP,Trans}$ can now be estimated. The first parameter $t_{OLAP,OH}$ is assumed to be constant, and is estimated by timing a probing query posed to the OLAP component. The probing query specifies a single measure and one particular combination of dimension values all belonging to bottom levels of the cube. Assuming that the cube has indexes on dimension values, the evaluation time of such a query is negligible. Also, the query returns at most one value, which means that little or no time is used on transporting data from the OLAP component. Hence, $t_{OLAP,OH}$ includes the full processing and network communication time of a query except the time it takes to actually produce the result and to transfer the result over the network. A better estimate can be achieved by using the average time for a number of queries.

The estimate of $t_{OLAP,Trans}$ for a query Q is calculated from the estimated result size and the network data transfer rate:

$$t_{OLAP,Trans} = \frac{Size(Q(C))}{NetworkDataRate(C)}$$

Both $Size(Q(C))$ and $NetworkDataRate(C)$ are estimated as described above. However, if the OLAP component is also used as temporary component, $t_{OLAP,Trans}$ is set to zero cost.

The evaluation cost $t_{OLAP,Eval}$ can be estimated in two ways. The simplest is to use the formula for $EvalTime$ directly and base all evaluation cost estimates on a single estimate of the cube size. Alternatively, the estimated cost for a query can be based on the measured cost for a similar query that has been posed earlier. The cost can be measured from a probing query or a user query. A list of these queries can be maintained together with their measured cost and used to compute the cost of future queries. The latter should intuitively provide the best estimates, since it is based on an actual measured cost for a similar query. This approach is described in more detail in the following.

Using the second method, $t_{OLAP,Eval}$ is estimated from a set of probing queries and the statistical information presented above. The probing queries are used to estimate the query evaluation time and result size of queries that aggregates the cube to a certain combination of levels without performing any selections. From this estimate the size and evaluation time of a given query can be calculated using the functions presented above. Queries that retrieve all data at a given combination of levels cannot generally be posed directly, and instead the size and evaluation time of such a query Q_{All} is estimated by posing a probing query Q_{Probe} as described later. These estimated results are stored for each cube C in a table containing a row for each query Q_{All}:

Columns	**Description**
$Dim_1, \ldots, Dim_n$	The combination of levels to which the cube is rolled up in Q_{All}
$Size(Q(C))$	The estimated size of the result of Q_{All}
$EvalTime_{with\ preagg}$	The estimated evaluation time of Q_{All} when preaggr. may have been used
$EvalTime_{without\ preagg}$	The estimated evaluation time of Q_{All} when preaggr. have not been used
$QueryCount$	The number of user queries that has rolled up to the same levels as Q_{All}
$PreDim_1, \ldots, PreDim_n$	The level of pre-aggregation that is assumed to be used to evaluate Q_{All}

Table 4: The statistics that are stored for OLAP queries.

Due to the typically large amounts of data stored in OLAP databases, it is often unfeasible to fetch the entire cube to get the size estimate and evaluation time estimates. This is especially true for the lower level aggregates, whereas the amount of data at higher aggregation levels can often be retrieved in its entirety. Instead, probing queries are used that select a certain percentage of the facts on the specified levels, that is, selections are performed on dimension values to keep the size of the returned result reasonable. Notice that if the simple size based method is used, it is sufficient to use probing queries like "SELECT COUNT(*) FROM F" to estimate the size. However, when using the second method the time must also be measured.

Example 10.1 Here is an example of a query that can be expected to select approximately 25% of the cube data, because all bottom values are selected in the ECs dimension, while 50% are selected in both the Suppliers and Time dimensions:

```
SELECT     SUM (Cost), SUM (NoOfUnits), Class(EC), Country(Supplier), Year(Day)
FROM       Purchases
WHERE      Class IN ('FF', 'L') AND Country IN ('US') AND Year(Day) IN ('2000')
GROUP BY   Class(EC), Country(Supplier), Year(Day)
```

Assuming a uniform distribution of facts in the cube this returns around 25% of the data. In a real example smaller amounts of data are used. □

Each probing query is timed resulting in $t_{Q_{Probe}}$, and the size of the result is measured, resulting in $Size_{Q_{Probe}}$. Again, assuming that facts are distributed uniformly in the cube, the actual size of the cube without the selections $Size_{Q_{All}}$ can then be estimated based on $Size_{Q_{Probe}}$ by using the $Size$ function mentioned in Table 3. Likewise, the evaluation time $EvalTime_{Q_{All}}$ can be estimated using the approximation $t_{Q_{Probe}} = t_{OLAP,OH} + EvalTime_{Q_{Probe}} + \frac{Size_{Q_{Probe}}}{NetworkDataRate(C)}$ and the definition of $EvalTime$. Two different values of the evaluation time are needed to provide good estimates for both queries that may make use of pre-aggregation and for queries that cannot use pre-aggregation, because some selection refers to unaggregated measures. To measure these values two probing queries are used: One that performs selection only on dimension values as described above, and one that also contains a selection on a measure in the WHERE clause. This forces the OLAP component to access the base cube directly, in effect disabling the use of pre-aggregated results. The problem with using the definition of $EvalTime$ is that it is not generally possible to know which pre-aggregations are used. Hence, an adaptive strategy is used as discussed in the presentation of $EvalTime$ above. However, if $EvalTime_{with\ preagg} \approx EvalTime_{without\ preagg}$, we can conclude that no pre-aggregations have been used even though the probing query permitted it. To support this adaptive strategy, the columns $PreDim_1, \ldots, PreDim_n$ contains the current guess at which level of pre-aggregation is used.

Example 10.2 Assume that a probing query has taken 5 sec. and returned 100 KB of data, and that the OLAP result is pre-aggregated. Also, $t_{OLAP,OH,Probe} = 1$ sec., $t_{OLAP,Trans,Probe} = 2$ sec., $DiskDataRate(C) = 10$ MB/s, and $S_L = 10\%$.

Then $t_{OLAP,Eval,Probe} = 5s - 1s - 2s = 2s$ which yields $t_{OLAP,Eval,All} = \frac{t_{OLAP,Eval,Probe}}{S_L} = \frac{2s}{0.1} = 20s$ Notice that this simple formula only holds because full pre-aggregation is assumed. The size can be estimated similarly: $Size(Q_{All}) = \frac{Size(Q_{Probe})}{S_L} = \frac{100KB}{0.1} = 1000KB$ □

Often, the number of different combinations of levels is high, which makes it unfeasible to execute a probing query for each combination. When a certain combination of levels is needed but is not present in the table, the $EvalTime$ function is used directly. The problem with this approach is that we do not know what pre-aggregated values have been used. However, here we can do better than to simply guess at some combination between full and no pre-aggregation, as is necessary if the estimation is based only on the cube size. The $PreDim_i$ columns in Table 4 provides information about which pre-aggregations are believed to have been used for lower level combinations. Hence, we base the guess on the best level of pre-aggregation occuring in the table instead of assuming no pre-aggregation as the worst case scenario. The table is updated such that the next time a query aggregating to the same level is posed, we can provide a better guess.

The statistics table can now be used to estimate the cost parameters $t_{OLAP,Eval}$ and $t_{OLAP,Trans}$ for an arbitrary query Q in the following way: First, determine to which levels the cube is rolled up in Q, and then retrieve the relevant values from the statistics table. If these values do not exist in the table, then they are computed as discussed above. Second, determine whether or not pre-aggregation can be used, and choose the correct value of $EvalTime$. Third, compute the new estimates of $Size$ and $EvalTime$, $Size_Q$ and $EvalTime_Q$, using the functions in Table 3 and the values in the statistics table. Finally, the two parameters can be estimated: $t_{OLAP,Eval} = EvalTime_Q$ and $t_{OLAP,Trans} = \frac{Size_Q}{NetworkDataRate(C)}$. After the query has been evaluated the statistics table is updated to reflect the actual cost as described next.

10.1.3 Maintaining the Statistical Information

The statistics table is updated for each query that is posed to the OLAP component. The assumed level of pre-aggregation, which is stored in the the $PreDim_i$ columns, is updated as described above, while the estimated values of $Size$ and $EvalTime$ are updated as follows.

For each user query, new $Size$ and $EvalTime$ values are computed for the corresponding Q_{All} query exactly as was described for probing queries. However, the measured costs may vary, e.g. because of network disturbances, and hence, the old value is not just replaced by the new value. Instead, a weighted average is used, based on the $QueryCount$ column: Let V_O, V_Q and V_N be the old value, the value obtained from the query, and the new value, respectively. Then $V_N = \frac{V_Q + V_O \cdot Min(MaxCount, QueryCount)}{Min(MaxCount, QueryCount)+1}$, where $MaxCount$ is a tuning parameter that ensures that the new value has a certain weight even when $QueryCount$ is large. Without this parameter, any changes to the cube, such as updates, would be reflected too slowly in the statistics table. The $QueryCount$ column is also used when determining which results should be pre-fetched, as those with a high count are most likely to result in a high hit-rate.

The statistics table for the OLAP component contains one row for each combination of levels that has been used in a query. If many different queries are posed over a long period of time this number may become large. In that case, the size can be kept at a fixed level by expiring old and less frequently used combinations each time a new row is added.

10.2 Determining Cost Parameters For the XML Components

Estimating cost for XML components is exceedingly difficult because little or nothing can be assumed about the underlying data source, i.e. XML components are non-conforming. An XML data source may be a simple text file used with an XPath engine [Pro01, Cla99], a relational [Cor01a, Cor01b] or OO [Sof01b, eC01] database or a specialized XML database [AG01a, Sof01a, GMW99]. The query optimization techniques used by these systems range from none at all to highly optimized. Optimizations are typically based on sophisticated indexing and cost analysis [MW99]. Hence, it is impossible e.g to estimate the amount of disk I/O required to evaluate a query, and consequently, only a rough cost estimate can be made. Providing a good cost model under these conditions is not the focus of this paper and hence, we describe only a simple cost model.

The cost model is primarily used to determine whether or not XML data should be inlined into OLAP queries. Hence, in general a pessimistic estimate is better than an optimistic, because the latter may cause XML data not to be inlined. This could result in a very long running OLAP query being accepted, simply because it is not estimated to take longer than the XML query. However, the actual cost will never be significantly larger than the false estimate. Making a pessimistic estimate will not cause this problem although it may sometimes increase the cost because XML data is retrieved before OLAP data instead of retrieving it concurrently. For that reason, conservative estimates are preferred in the model.

The model presented here is based on estimating the amount of data returned by a query, and assuming a constant data rate when retrieving data from the component. Similar to the cost formula for OLAP queries, we distinguish between the constant overhead of performing a query $t_{XML,OH}$ and the time it takes to actually process the query $t_{XML,Proc}$:

$$t_{XML} = t_{XML,OH} + t_{XML,Proc}$$

Hence, only the latter depends on the size of the result.

In the following we describe how to estimate these two cost parameters given an XPath query. Although other more powerful languages may be used, the estimation technique can easily be changed to reflect this. For simplicity we consider only a subset of the XPath language where XPath expressions are on the form described in Section 9.3. Because XML components are non-conforming, the estimates are based on posing probing queries to the XML component to retrieve the necessary statistics.

10.2.1 Statistical Information

Estimation of the cost parameters $t_{XML,OH}$ and $t_{XML,Proc}$ is based on the statistical information described in the following. The way this information is obtained and used to calculate the cost parameters is described later. In the descriptions N is the name of a node, e.g. the element node "Supplier" or the attribute node "NoOfUnits", while E denotes the name of an element node. Let $path_{E_n}$ be a simple XPath expression on the form $/E_1/E_2/\ldots/E_n$, that specifies a single direct path from the root to a set of element nodes at some level n without applying any predicates.

$NodeSize(path_E)$**:** The average size in bytes of the nodes pointed to by $path_E$. The size of a node is the total size of all its children, if any.

$Fanout(path_{E_n})$**:** The average number of E_n elements pointed to by each of its parent elements E_{n-1}. Notice that there may be E_n elements that are children of other elements, since there can be several paths to the same type of element. The fanout is estimated for each of these paths.

$Selectivity(\theta)$**:** The selectivity of predicate θ in its given context. For simplicity, two types of predicates are distinguished: Simple predicates and complex predicates. Simple predicates are on the form $x_1 \otimes x_2$, where each x_i is either a node with a numeric content or a numeric constant and $\otimes$ is a comparison operator. The selectivity of these predicates are estimated from the maximum and minimum values as decribed for OLAP queries. All other predicates are complex and may refer to non-numeric nodes and various functions, e.g. for string manipulation. (An exception is predicates involving the $position()$ function, which is estimated as a simple predicate.) Following previous work [BMG93], the selectivity of complex predicates is set to a constant value of 10%.

$Cardinality(path_{E_n})$**:** The total number of elements pointed to by $path_{E_n}$. The cardinality of E_n can be calculated from any ancestor element E_k along the path using this formula:

$$Cardinality(path_{E_n}) = Cardinality(path_{E_k}) \cdot \prod_{i=k+1}^{n} Fanout(path_{E_i}) \cdot Selectivity(\theta_i)$$

If no predicate occurs in an element E_i the selectivity of θ_i is 100%.

$DataRate(x)$**:** The average amount of data that can be retrieved from the XML document x per second. Given a set of queries $xp_1, \ldots, xp_n$ the data rate can be estimated like this:

$$DataRate(x) = \frac{\sum_{i=1}^{n}(time(xp_i) - t_{XML,OH})}{\sum_{i=1}^{n} size(xp_i)}$$

where $time(xp)$ and $size(xp)$ gives the total evaluation time and result size of query xp, respectively.

As a refinement, this data rate can be estimated on a per element basis, which may give better performance for some types of components.

10.2.2 Obtaining and Using Statistical Information

Some of the information discussed above can be obtained directly if an XML Schema is available [W3C01a]. In that case, information such as *NodeSize*, *Cardinality*, or *Fanout* is determined from the schema. DTDs [W3C00] can also be used to provide this information, but only if no repetition operators (i.e. the "*" and "+" operators) are used. However, if such meta data is not available, then the statistical information is obtained using probing queries that are based on the links defined for each XML document. Which queries can be used depends on the type of the link. Three different types of probing queries are used as shown here:

No.	Query	Measured values
Probe1	/$*$[false()]	time
Probe2	/E$_1$[position()=i$_1$]/ ... /E$_k$[position()=i$_n$]	fanout, cardinality
Probe3	/base[locator=v_i] (Natural links) /locator$_{v_i}$ (Enumerated links) for n random values of i	time, size, fanout, cardinality, min value, max value

From Probe1 the query overhead $t_{XML,OH}$ can be estimated. The assumptions are that no data is returned and that the work performed by this query will always have to be performed. Hence, it represents the minimum time it takes to perform a query. This may include everything from parsing the entire XML document to parsing only the query. In either case it is a reasonable approximation to the constant overhead, which will often be dominated by the server response time. However, sometimes a clever parser may discover that this query always produces an empty result and avoid most of the query overhead. In that situation, requesting a leaf node would usually provide a better estimate of the overhead. By running the probing query a number of times a more precise average value can be found.

A number of queries on the form of Probe2 are used to find the fanout and cardinality of the elements above and including the nodes pointed to by the link. For smaller amounts of XML data, this is done by simply retrieving all the nodes. However, for large amounts of data this may not be feasible, and another method must be used. Since XPath does not allow computed values such as count() to be returned, binary search is used to find the maximum value i_j of position() for which any data is returned. The idea is to find the number of values on the first level, use a sample of these to estimate the number of values on the second level, use a sample of these to estimate the number of values on the third level, and so on. Most of these queries will work on data that has already been retrieved, and hence, only a few queries will actually retrieve data. However, these queries may return large amounts of data because they refer to nodes close to the root node. If this is not feasible, a guess is made at the number of nodes.

Probe3 is used to find statistical information about the nodes below the nodes pointed to by the link. A sample of the nodes pointed to is retrieved using Probe3 for the given type of link. The nodes are then analyzed locally to find the remaining information: $Fanout$, $Cardinality$ and $Size$ of the remaining elements, as well as maximum and minimum values for numerical nodes. The time and size of the queries are measured and used in the computation of $DataRate$. Where several links exist for a single document, only one set of probing queries is performed. All the statistical information obtained from the probing queries is stored in the federation metadata for each XML document.

The combination of a limited interface and almost no knowledge about the underlying source can sometimes make the probing of XML components expensive. As discussed in Section 9.3 several optimizations are possible, including the retrieval of larger parts of the document in a single query. However, probing may still be a problem for very slow XML components, e.g. on the Web. Hence, the default behaviour is not to perform probing immediately, when a link is created, but instead wait until the system load has been low for a while. When no probing has been performed, nothing is known about the component, and instead a predefined set of cost values are used. The probing can also be disabled for very slow components.

The cost of an XPath expression can now be computed using the cost formula for XML queries. $t_{XML,OH}$ is estimated as described above, while $t_{XML,Proc}$ is calculated for an XPath expression xp in a document X as follows:

$$t_{XML,Proc} = \frac{NodeSize(xp) \cdot Cardinality(xp)}{DataRate(x)}$$

Example 10.3 Let $t_{XML,OH} = 1$ s, $DataRate = 32$ KB/s, $NodeSize = 120$ Bytes, and $Cardinality = 500$. Then the total cost can be calculated as:

$$t_{XML} = 1s + \frac{120Bytes \cdot 500}{32KB/s} \approx 2.8s$$

□

An approach similar to that for OLAP queries is used to keep the statistical information updated. Hence, when new XML data has been retrieved, it is analyzed and the values for fanout, cardinality, node size, and minimum and maximum values are corrected. This analysis is performed only when the system load is low. Hence, overall performance is not affected.

10.3 Determining Cost Parameters for the Temporary Component

The temporary relational component is used as a scratch-pad by the Federation Manager during the processing of a query. For efficiency, we assume to have full access to its meta data, such as cardinalities, attribute domains and histograms, as well as knowledge about which algorithms are implemented for processing operations. Additionally, we have full knowledge about which access paths are available because all tables are created by the federation itself. Hence, the temporary component is assumed to be a conforming component. Also, for simplicity we assume that the temporary component is located on the same node as the Federation Manager. This allows us to ignore network costs for this component. Consequently, existing work on query optimization can be used to provide efficient access to this component. Specifically, we use a simplified variant of the cost model defined in [DKS92]. This model has been demonstrated to provide good results for different DMBSs.

11 Overview of the Federated System

In this section we give an overview of the presented design considerations and optimization techniques as well as their use in the federated system.

An overall architectural design of a prototype system supporting the SQL_{XM} query language has been presented in Figure 8. Besides the OLAP component and the XML components, three auxiliary components have been introduced to hold meta data, link data, and temporary data. Generally, current OLAP systems either do not support non-strict hierarchies or it is too expensive to add a new dimension, which necessitates the use of a temporary component. SQL_{XM} queries are posed to the *Federation Manager*, which coordinates the execution of queries in the data components using several optimization techniques to improve query performance. Since the primary bottleneck in the federation will usually be the moving of data from OLAP and XML components, our optimization efforts have focused on this issue. These efforts include both *rule based* and *cost based* optimization techniques, which are based on the transformation rules for the federation algebra.

The *rule based* optimization uses the heuristic of pushing as much of the query evaluation towards the components as possible. Although not generally valid, this heuristic is useful in our case, since the considered operations all reduce the size of the result obtained from the data components. The rule based optimization algorithm that has been presented *partitions* a SQL_{XM} query tree, meaning that the SQL_{XM} operators are grouped into an OLAP part, an XML part, and a relational part. After partitioning the query tree, it has been identified to which levels the OLAP component can be aggregated and which selections can be performed in the OLAP component. Furthermore, the partitioned query tree has a structure that makes it easy to create component queries.

Three different *cost based* optimization techniques have been presented: One targeted at OLAP components, one at XML components and a more general one targeted at both types of components. Together, these techniques offer a considerable performance improvement in general and in particular for many queries that would otherwise be too costly.

The first technique tries to tackle one of the fundamental problems with the idea of evaluating part of the query in a temporary component: If selections refer to data that are not present in the result, more than the result needs to be transferred to the temporary component. The proposed solution to this problem is to *inline* XML data into OLAP predicates. However, it is not always a good idea to do so because, in general, a single query cannot be of arbitrary length. Hence, more than one query may have to be used. Whether or not XML data should be inlined into some OLAP query, is decided by comparing the estimated cost of inlining with the estimated cost of not doing so.

The second technique is focused on the special kind of XML queries that are used in the federation. These queries can easily be expressed in more powerful languages like XQuery, but many XML sources have more limited interfaces, such as XPath or XQL. The special queries needed to retrieve data from XML components cannot be expressed in a single query in these simple languages, and hence, special techniques must be used for this to be practical. The main solution suggested here is to combine these queries, even though more data would have to be retrieved. Again, a cost analysis is used to decide whether or not to employ this technique.

The third technique is an application of *caching* to this particular domain. The use of caching is important for both OLAP and XML components, as both types of components may cause significant delays for certain kinds of queries. One of the approaches that was presented here is an efficient way to find a useful cached result for a given OLAP query. *Pre-fetching* has also been suggested as a way to speed up queries that have not been posed before.

The use of cost based optimization requires the estimation of several cost parameters. One of the main arguments for federated systems is that components can still operate independently from the federation. However, this autonomy also means that little cost information will typically be available to the federation. Hence, providing a good general cost model is exceedingly difficult. In this context, it is especially true for XML components, because of the wide variety of underlying systems that may be found. Two general techniques have been used to deal with these problems: *Probing queries*, which are used to collect cost information from components, and *adaption*, which ensures that this cost information is updated when user queries are posed.

To outline how the techniques discussed above are used in combination, we will refer to the software component architecture in Figure 9. When a federation query has been parsed, the Query Decomposer partitions the resulting SQL_{XM} query tree, splitting it into three parts: an OLAP query, a relational query, and a number of XML queries. The XML queries are immediately passed on to the Execution Engine, which determines for each query whether the result is obtainable from the cache. If this is not the case, it is sent to the Component Query Evaluator. Here, the specific actions depend on which query languages are supported. If e.g. only an XPath interface is available, it is determined which is cheaper: To pose many queries or to combine some or all of them into a single query. This decision is based on costs estimated by the Component Cost Evaluator. For the OLAP part of the query, it is also determined whether the result can be obtained from any of the cached results. If so, the cost of evaluating the SQL_{XM} query using the cached results is compared to the cost of evaluating the SQL_{XM} query without the use of cached results. This cost is determined by the Global Cost Evaluator. The cost estimates are also used by the Global Optimizer to pick a good inlining strategy. When the results of the component queries are available in the temporary component, the relational part of the SQL_{XM} query is evaluated.

In the next section we describe the implementation of the federation prototype. A number of experiments have been conducted which are also presented.

12 Implementation and Experimental Results

This section describes the ongoing prototype implementation of the architecture presented in Section 7. Some initial experiments are also reported.

A prototype is currently being developed based on the overall and software component architecture presented in Figure 8 and 9, respectively. In the prototype, the OLAP component uses Microsoft Analysis Services and is queried with MDX and SQL[Mic98]. The XML component is based on Software AG's Tamino XML Database system [AG01a], which provides an XPath-like interface. As discussed earlier, the possibilities for optimization depend on the degree of autonomy exhibited by components. Thus, we have performed experiments for two different settings: First, the general case where the OLAP component has a high degree of autonomy and an external component is used for temporary data. Second, the highly optimized situation where the OLAP component is also used for temporary data and all XML data is assumed to be cached. For the external temporary component, a single Oracle 8*i* system is used. The prototype is being implemented in Java using

Microsoft ADO and ADO-MD APIs following the component architecture discussed in Section 8.1. So far, only the most basic functionality has been implemented. In particular, the cost based optimization techniques have not been implemented yet. Nevertheless, it has been possible to conduct initial experiments that support the idea of a federation as a practical alternative to physical integration for rapidly changing environments.

The example cube used in the experiments is shown in Figure 15. The cube is based on about 50 MB of data generated using the TPC-H benchmark [Cou01] and about 10 MB of pre-aggregated results. The cache size was limited to 10 MB to prevent an unrealistically large part of the data from being cached.

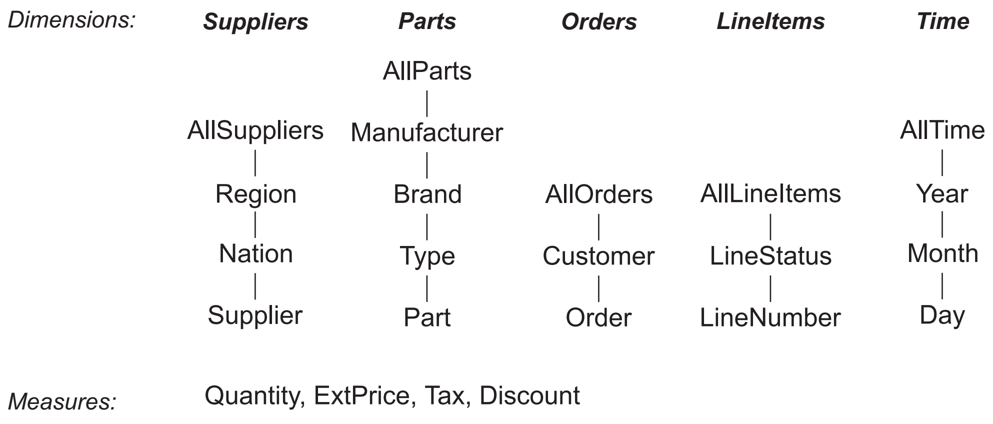


Figure 15: Example cube used in the experiments.

About 3 MB of XML data is used for the XML component which is divided into two documents that have been generated from the TPC-H data and public data about nations. The structure of these documents is illustrated in Figure 16. Two natural links are defined: NLink = (“Nation”, “nations.xml”, “/Nations/Nation/”, “NationName”) and TLink = (“Type”, “types.xml”, “/Types/Type/”, “TypeName”). The nodes used for decoration are Population and RetailPrice.

```
<Nations>
  <Nation><NationName>Denmark</NationName><Population>5.3</Population></Nation>
  <Nation><NationName>China</NationName><Population>1264.5</Population></Nation>
  <Nation><NationName>Mozambique</NationName><Population>19.1</Population></Nation>
  ...
</Nations>

<Types>
  <Type><TypeName>Promo Polished Brass</TypeName><RetailPrice>1890</RetailPrice></Type>
  <Type><TypeName>Promo Burnished Copper</TypeName><RetailPrice>1240</RetailPrice></Type>
  <Type><TypeName>Medium Brushed Steel</TypeName><RetailPrice>1410</RetailPrice></Type>
  ...
</Types>
```

Figure 16: Part of the two XML documents used in the experiments.

The queries that were used in the experiments are shown in Table 5 and the results of these queries are shown in Figure 17. Result 1-7 are used to illustrate the situation where the OLAP component has a high degree of autonomy, whereas Result 8 compares these results to the highly optimized situation where all data is stored in the OLAP component. The results do not show the overhead of parsing and optimizing the federation queries but only the component evaluation times as these will dominate the total evaluation time. In the results, the total evaluation time for each query is divided into three parts: One for each of the three types of component queries posed during the evaluation of a federation query. Thus, the following tasks are represented in the results:

Task 1(a). Fetch XML data and store it in the temporary component. (Denoted by an “X”)

Task 1(b). Fetch OLAP data and store it in the temporary component. (Denoted by an “O”)

Task 2. Compute the final result in the temporary component. (Denoted by a “T”)

Task 1(a) and 1(b) can be performed in parallel unless the XML data is inlined into the OLAP query. We first consider the setting where an autonomous OLAP component is used and thus, all three tasks have to be performed.

Label	**Query**

A_1	SELECT SUM(Quantity), SUM(ExtPrice), Nation(Supplier), Brand(Part), LineStatus(LineNumber), Nation/NLink/Population
	FROM Sales
	GROUP BY Nation(Supplier), Brand(Part), LineStatus(LineNumber), Nation/NLink/Population

A_2	SELECT SUM(Quantity), SUM(ExtPrice), Nation(Supplier), Brand(Part), Nation/NLink/Population
	FROM Sales
	GROUP BY Nation(Supplier), Brand(Part), Nation/NLink/Population

B	SELECT SUM(Quantity), SUM(ExtPrice), Brand(Part), LineStatus(LineNumber), Nation/NLink/Population
	FROM Sales
	GROUP BY Brand(Part), LineStatus(LineNumber), Nation/NLink/Population

C	SELECT SUM(Quantity), SUM(ExtPrice), Nation(Supplier), Brand(Part), LineStatus(LineNumber),
	FROM Sales
	GROUP BY Nation(Supplier), Brand(Part), LineStatus(LineNumber)
	HAVING Nation/NLink/Population > 10

D (1-3)	SELECT SUM(Quantity), SUM(ExtPrice), Type(Part)
	FROM Sales
	WHERE Nation/NLink/Population > 10 AND Type/TLink/RetailPrice < 1000
	GROUP BY Type(Part)

Table 5: Queries used in the experiments.

Query A_1 and A_2 both illustrate the use of decoration. The results of these queries are shown in Result 1. Furthermore, they illustrate the basic structure of the cost model presented in Section 9.1 as the temporary component query must wait for the slower of the XML and OLAP queries. Result 1 displays a low overhead of performing decoration compared to the time it takes to retrieve the OLAP and XML data. Thus, if it is acceptable to wait for the component data when retrieved independently, it will also be acceptable to wait for the federation query. This low overhead is representative for decoration queries since they do not require additional data to be retrieved from the OLAP component. Hence, the overhead of combining the intermediate results will be low, since typically only small amounts of OLAP data (at most a few thousand facts) will be requested for presentation to a user.

Query B illustrates the use of XML data for grouping, while Query C uses XML data for selection. Since both queries use the same decoration as in Query A_1, the same XML and OLAP queries are performed to evaluate these queries. As can be seen from Result 2 and 3, the overhead endured by the temporary component query is also low for these queries. This is typical for both grouping and certain types of selection because the size of the intermediate OLAP result will often be comparable to the size of the final result. Again, since the final result is mostly presented to a user, it is often relatively small. For grouping, the OLAP and final results are comparable in size unless there is a great overlap in the decoration values which reduces the size of the final result. In this case, Query B has approximately as many Population values as there are Nation values. For selections, the OLAP result is often comparable in size to the final result for queries such as Query C, where the predicate refers to decorations of levels that must be present in the result. This is true unless the predicate is very selective. However, as discussed in Section 9, selections may also refer to lower levels. To handle the

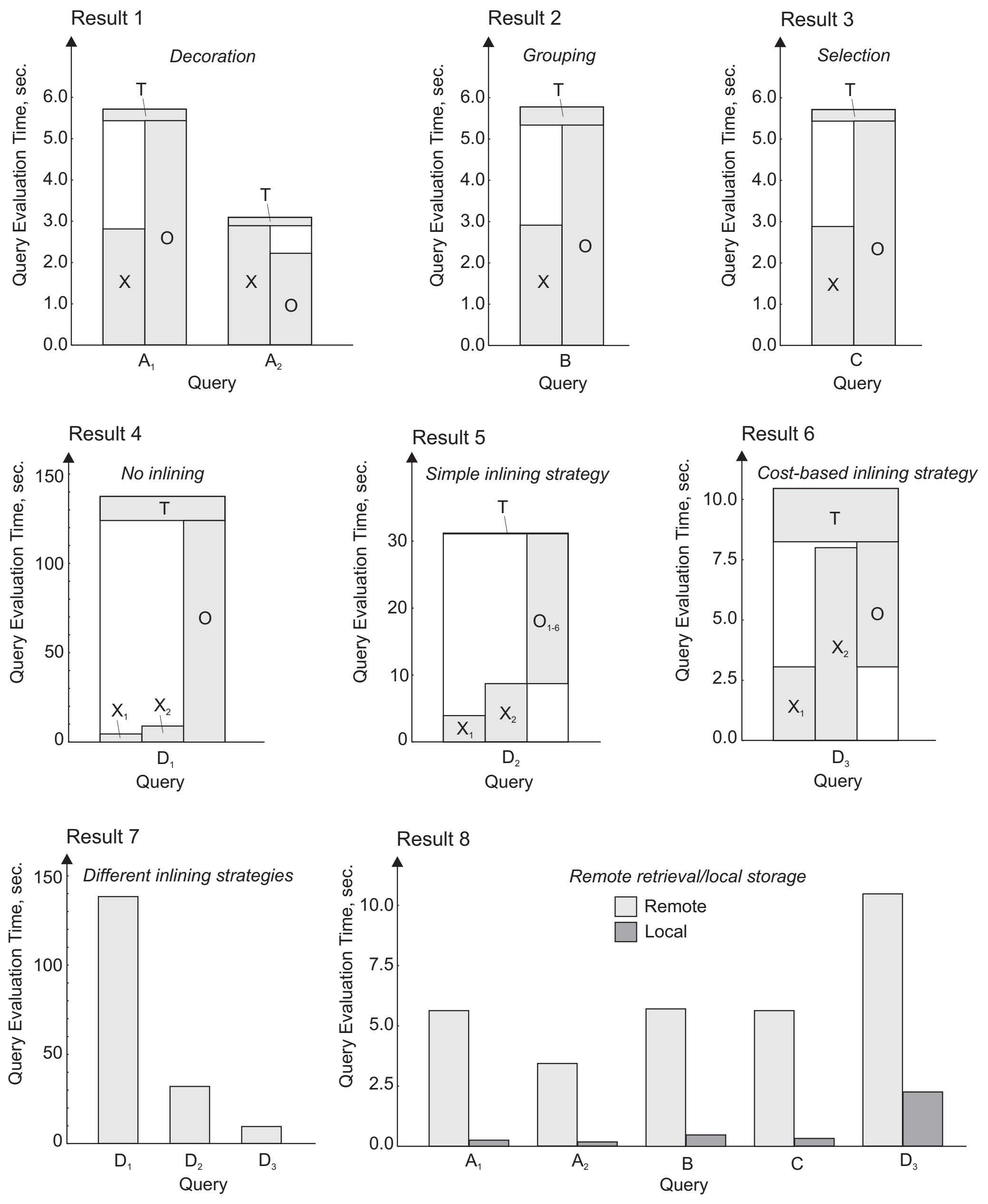


Figure 17: Experimental results. Notice that different time intervals are used in the graphs.

problem of selections requiring large amounts of data to be retrieved, the inlining technique was presented. This technique has not been used in evaluating Query C.

Results 4, 5, and 6 demonstrate how effective the inlining technique can be. Here, two XML queries are used: X_1 is the time it takes to retrieve the Population data, while X_2 is the time it takes to retrieve the RetailPrice data. The results show the processing times for three different strategies for evaluating Query D, in which the selection on Population refers to a low level that is not needed in the result. Thus, in Result 4, where no inlining is used, this produces a very large overhead as the OLAP query can only aggregate to Type and Nation although only Type should be present in the result. As a consequence, not only the OLAP query but also the temporary component query take much longer to evaluate. Result 5 illustrates the use of a simple rule based inlining strategy, “Always use inlining if the predicate is simple”, which causes both XML predicates to be inlined. This is significantly faster than before because the OLAP query can now aggregate to the Type level. Also, since the WHERE clause has been evaluated entirely in the cube, no work needs to be done after the OLAP result has been returned. However, six OLAP queries are needed to hold the new predicate because the Type level contains a large number of dimension values. Also, the OLAP query cannot be evaluated until the RetailPrice data has been retrieved. Consequently, in this example, it is faster to inline only the Population data. This is shown in Result 6 where the cost based inlining strategy presented in Section 9.2 is used. Because the Nation level only contains few dimension values, this requires only one OLAP query and the Population data is also faster to retrieve. Thus, by estimating the cost of the federation query for each of the four inlining strategies and picking the fastest one, a better query performance is achieved. The three results are also shown in Result 7 for easy comparison. These results have shown examples of the different types of queries supported by the federation as well as illustrated the potentially large performance improvements possible by using the inlining technique. These evaluation times are acceptable even though the component data is retrieved from a remote data source and transferred to the temporary component.

If the OLAP component does not have a high degree of autonomy, e.g. if it is managed by the federation DBA, large performance improvements are possible by using caching and by storing temporary data with the OLAP data. By caching or pre-fetching XML data and storing it in tables, Task 1(a) can be avoided. Caching XML data is often possible as the amount of XML data will typically be much smaller than the amount of OLAP data. If only few XML documents are used, it may even be possible to keep the cache updated by pre-fetching XML data at a regular basis. If a ROLAP (Relational OLAP) system is used for storing OLAP data, all pre-aggregated results are stored directly in relational tables, whereas if a MOLAP (Multidimensional OLAP) or HOLAP (Hybrid OLAP) system is used, some additional processing is needed to produce these tables. Thus, if the ROLAP system can also be used for storing temporary data, the tables containing pre-aggregated results can be joined directly with the tables containing XML data. In that case, Task 1(b) can be avoided. This is not an unreasonable assumption as the OLAP DBA will often also be responsible for managing the federation and thus, be able to use the OLAP component to store the temporary data. Hence, if both cached XML data and pre-aggregated results are available as relational tables, only Task 2 is necessary. To illustrate the potential performance improvements in this situation, all queries in Table 5 have also been evaluated using a fully pre-aggregated cube where both these OLAP data and XML data were stored in tables. These *local* evaluation times are shown in Result 8 together with the non-local, or *remote*, evaluation times described above. As expected, the local values are significantly smaller than the remote values. Since these local evaluation times require a low degree of autonomy for the OLAP component, physically integrating the data is also an option. If XML data is physically integrated into the cube and the resulting cube is fully pre-aggregated, no processing is needed as the result can be read directly from disk. However, this requires additional storage compared to the federated approach as the new cube contains more dimensions. If such storage is not available, a smaller percentage of the new cube can be pre-aggregated compared to the original cube. Thus, on average, an OLAP query working on the new cube will have to perform more aggregation than an OLAP query working on the original cube. More significant is the more or less manual work that needs to be done to integrate the data, as well as the time it takes to re-process the cube.

The results presented in this section indicate that for many useful queries the federated approach is indeed a practical alternative to physical integration when flexibility is needed. For the analyzed queries, the federation overhead is insignificant compared to the time it takes to retrieve data from the OLAP and XML components. As is the case for any database system, including ordinary OLAP systems, there are also queries that cannot be evaluated efficiently. However, the optimization techniques presented in this paper can provide significant performance improvements for many federation queries, in effect allowing otherwise unfeasible queries to be evaluated efficiently. In particular, the potentially dramatic effect of the caching and inlining techniques have been demonstrated in this section. Further experiments are required to make any decisive conclusions about the general performance of our federated approach compared to alternative solutions, such as physical integration.

13 Conclusion

Many OLAP systems operate in dynamic business environments where changes to data and data requirements are common and data may not always be available for local storage but only for querying. In these situations physical integration is not feasible, making logical integration in a federation the better choice.

In this paper we have presented a flexible approach to the logical federation of OLAP databases and XML documents. This allows XML data to be used directly in an OLAP query to decorate multidimensional cubes with XML data from external sources, to group cube data based on external XML data, and to perform selection based on external XML data. The incorporation of XML data in a cube is made such that semantic problems are avoided, e.g. aggregation types are used to ensure that no measure values are double-counted when aggregation is performed.

The theoretical and practical work on this federated approach has covered all important aspects of a federated system. The fundamental idea was formally defined in terms of a federated data model and algebraic query language. As a demonstration of this general formal approach, the algebra was used to define the semantics of a federated query language, SQL_{XM}. This language incorporates the XML query language XPath into a subset of SQL adapted to multidimensional querying. A complete federated system supporting this language was designed and a number of effective *rule based* and *cost based* query optimization techniques were presented. To provide a basis for these optimizations, a collection of *transformation rules* was presented for the federation algebra. The rule based optimization distributes query processing on the participating components using the presented *partitioning algorithm*. The cost based optimization focuses on three techniques: First, *inlining* of XML data was suggested to reduce the overhead of evaluating queries that uses external XML data for selection. Second, for simple languages, such as XPath, the special queries needed to retrieve data from XML components can only be expressed using several queries. Hence, techniques were described to combine these queries at the cost of retrieving additional data. Third, techniques were presented to allow *caching* and *pre-fetching* of intermediate query results. These techniques provide dramatic performance improvements for many queries that would otherwise be unfeasible to evaluate in the federated system. To illustrate the practical value of the system, a prototype is being developed, and the implementation has progressed far enough to allow experiments to be conducted. These indicate that this federated approach is indeed a practical alternative to physical integration when additional flexibility is needed.

We believe that this paper is the first to consider the federation of OLAP databases and XML documents. Specifically, we believe to be the first to consider advanced OLAP issues such as dimension hierarchies, automatic aggregation, and correct aggregation of data in the context of integration with XML data. Additionally, we extend previous work on federating OLAP databases with external data significantly by relaxing requirements for the data used for integration. Also, we propose a more general integration approach that can be used directly for selection and grouping. This approach is based on the *decoration operation* which has been formally defined and analyzed with respect to its equivalence properties. Also, query optimization issues for federations involving OLAP databases have not been investigated to this extent before.

Our immediate future work will focus on implementation aspects. In particular, the current prototype needs additional work, but it could also be interesting to explore how this work could be used in a real software product. For example, the ability to quickly integrate XML data could be incorporated into an existing OLAP querying tool. Here, a key issue which has not been considered yet, would be the design of a user tool to aid in the process of linking XML and OLAP data. Other interesting research issues include how to capture the document order of an XML document in the result of an OLAP query, and how to extract new measures from XML data and incorporate these into a cube. Also, other query languages than SQL and XPath could be considered for the federation. For example, it could be investigated how to integrate XPath expressions into the OLAP query language MDX without violating its syntax. Another area that needs more work is how to provide better cost estimates when querying autonomous OLAP components and, in particular, autonomous XML components.

References

[AG01a] Software AG. Tamino XML Database. `http://www.softwareag.com/taminoplatform`, 2001. Current as of June 15, 2001.

[AG01b] Software AG. Tamino XML Database FAQ. `http://www.softwareag.com/tamino/services/faq.htm`, 2001. Current as of June 15, 2001.

[AGS97] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling Multidimensional Databases. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pp. 232–243, 1997.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, volume 29, pp. 261–272, 2000.

[BC00] Angela Bonifati and Stefano Ceri. Comparative Analysis of Five XML Query Languages. *SIGMOD Record*, 29(1):68–79, 2000.

[BKLW99] Susanne Busse, Ralf-Detlef Kutsche, Ulf Leser, and Herbert Weber. Federated Information Systems: Concepts, Terminology and Architectures. Technical Report Forschungsberichte des Fachbereichs Informatik 99-9, 1999.

[BMG93] José A. Blakeley, William J. McKenna, and Goetz Graefe. Experiences Building the Open OODB Query Optimizer. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 287–296, 1993.

[CCS00] Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. On Wrapping Query Languages and Efficient XML Integration. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 141–152, 2000.

[CGMH$^+$94] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS Project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, pp. 7–18, 1994.

[Cla99] James Clark. XT. `http://www.jclark.com/xml/xt.html`, 1999. Current as of June 15, 2001.

[Cor01a] Microsoft Corporation. SQL Server. `http://msdn.microsoft.com/sqlserver/Default.asp`, 2001. Current as of June 15, 2001.

[Cor01b] Oracle Corporation. Oracle: XML Enabled. `http://www.oracle.com/xml`, 2001. Current as of June 15, 2001.

[Cou01] Transaction Processing Performance Council. TPC-H. `http://www.tpc.org/tpch`, 2001. Current as of June 15, 2001.

[CRF00] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceedings of the Third International Workshop on the Web and Databases*, pp. 53–62, 2000.

[CT98] Luca Cabibbo and Riccardo Torlone. A Logical Approach to Multidimensional Databases. In *Proceedings of the 6th International Conference on Extending Database Technology*, pp. 183–197, 1998.

[Dat01] DataJoiner. `http://www-4.ibm.com/software/data/datajoiner`, 2001. Current as of June 15, 2001.

[DD99] Ruxandra Domenig and Klaus R. Dittrich. An Overview and Classification of Mediated Query Systems. *SIGMOD Record*, 28(3):63–72, 1999.

[DKS92] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. Query Optimization in a Heterogeneous DBMS. In *18th International Conference on Very Large Data Bases*, pp. 277–291, 1992.

[eC01] eXcelon Corporation. eXcelon. `http://www.exceloncorp.com`, 2001. Current as of June 15, 2001.

[Eci01] ECIX QuickData Architecture. `http://www.si2.org/ecix`, 2001. Current as of June 15, 2001.

[EN00] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, third edition, 2000.

[Gat01] Gateways. `http://www.oracle.com/gateways`, 2001. Current as of June 15, 2001.

[GBC98] Goetz Graefe, Ross Bunker, and Shaun Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. In *Proceedings of 24rd International Conference on Very Large Data Bases*, pp. 86–97, 1998.

[GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Proceedings of the Twelfth International Conference on Data Engineering*, pp. 152–159, 1996.

[GdB94] Paul W. P. J. Grefen and Rolf A. de By. A Multi-Set Extended Relational Algebra - A Formal Approach to a Practical Issue. In *ICDE*, pp. 80–88, 1994.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proceedings of 21th International Conference on Very Large Data Bases*, pp. 358–369, 1995.

[GL98] Frédéric Gingras and Laks V. S. Lakshmanan. nD-SQL: A Multi-Dimensional Language for Interoperability and OLAP. In *Proceedings of 24rd International Conference on Very Large Data Bases*, pp. 134–145, 1998.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Workshop on the Web and Databases (WebDB '99)*, pp. 25–30, 1999.

[GMY99] Hector Garcia-Molina and Ramana Yerneni. Coping with Limited Capabilities of Sources. In *8th GI Fachtagung: Datenbanksysteme in Buero, Technik und Wissenschaft*, pp. 1–19, 1999.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GW00] Roy Goldman and Jennifer Widom. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 285–296, 2000.

[HSC99] Joseph M. Hellerstein, Michael Stonebraker, and Rick Caccia. Independent, Open Enterprose Data Integration. *IEEE Data Engineering Bulletin*, 22(1):43–49, 1999.

[Hsi92] David K. Hsiao. Federated Databases and Systems: Part I - A Tutorial on Their Data Sharing. *VLDB Journal*, 1(1):127–179, 1992.

[IW87] Yannis E. Ioannidis and Eugene Wong. Query Optimization by Simulated Annealing. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference,*, pp. 9–22, 1987.

[KGV83] S. Kirpatrick, C.D. Gelatt, Jr., and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, May 1983.

[Kim96] Ralph Kimball. *The Data Warehouse Toolkit : Practical Techniques for Building Dimensional Data Warehouses*. John Wiley & Sons, 1996.

[LAW99] Tirthankar Lahiri, Serge Abiteboul, and Jennifer Widom. Ozone - Integrating Semistructured and Structured Data. *The Eighth International Workshop on Database Programming Languages*, 1999.

[Leh98] Wolfgang Lehner. Modelling Large Scale OLAP Scenarios. In *Proceedings of the 6th International Conference on Extending Database Technology*, pp. 153–167, 1998.

[LS97] Hans-J. Lenz and Arie Shoshani. Summarizability in OLAP and Statistical Data Bases. In *Proceedings of the Ninth International Conference on Statistical and Scientific Database Management*, pp. 39–48, 1997.

[LTD95] Hongjun Lu, Kian-Lee Tan, and Son Dao. The Fittest Survives: An Adaptive Approach to Query Optimization. In *Proceedings of 21th International Conference on Very Large Data Bases*, pp. 251–262, 1995.

[Mic98] Microsoft Corporation. *OLE DB for OLAP Version 1.0 Specification*, 1998. Microsoft Technical Document.

[MW99] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proceedings of 25th International Conference on Very Large Data Bases*, pp. 315–326, 1999.

[PJ99] Torben Bach Pedersen and Christian S. Jensen. Multidimensional Data Modeling for Complex Data. In *Proceedings of the Fifteenth International Conference on Data Engineering*, pp. 336–345, 1999.

[PJD99] Torben Bach Pedersen, Christian S. Jensen, and Curtis E. Dyreson. Extending Practical Pre-Aggregation in On-Line Analytical Processing. In *Proceedings of 25th International Conference on Very Large Data Bases*, pp. 663–674, 1999.

[Pro01] The Apache XML Project. Xalan. `http://xml.apache.org/xalan`, 2001. Current as of June 15, 2001.

[PSGJ00] Torben Bach Pedersen, Arie Shoshani, Jun Min Gu, and Christian S. Jensen. Extending OLAP Querying to External Object Databases. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, pp. 405–413, 2000.

[RAH$^+$96] Mary Tork Roth, Manish Arya, Laura M. Haas, Michael J. Carey, William F. Cody, Ronald Fagin, Peter M. Schwarz, Joachim Thomas II, and Edward L. Wimmers. The Garlic Project. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, page 557, 1996.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). `http://www.w3.org/TandS/QL/QL98/pp/xql.html`, 1998. Current as of June 15, 2001.

[ROH99] Mary Tork Roth, Fatma Ozcan, and Laura M. Haas. Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System. In *Proceedings of 25th International Conference on Very Large Data Bases*, pp. 599–610, 1999.

[RR83] Maurizio Rafanelli and Fabrizio L. Ricci. Proposal of a Logical Model for Statistical Data Base. In *Proceedings of the Second International Workshop on Statistical Database Management*, pp. 264–272, 1983.

[RS90] Maurizio Rafanelli and Arie Shoshani. STORM: A Statistical Object Representation Model. In *Proceedings of the 5th International Conference SSDBM*, pp. 14–29, 1990.

[SDNR96] Amit Shukla, Prasad Deshpande, Jeffrey F. Naughton, and Karthikeyan Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. In *Proceedings of 22th International Conference on Very Large Data Bases*, pp. 522–531, 1996.

[SL90] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[Sof01a] IXIA Soft. TEXTML Server. `http://www.textmlserver.com`, 2001. Current as of June 15, 2001.

[Sof01b] Ascential Software. Axielle. `http://www.ascentialsoftware.com/solutions/axielle/additionInfo/axielle%_prodinfo.htm`, 2001. Current as of June 15, 2001.

[SSV98] S. Schelstraete, W. Schepens, and H. Verschelde. Energy minimization by smoothing techniques: a survey. In *Molecular Dynamics. From Classical to Quantum Methods*, pp. 129–185, 1998.

[Tho97] Erik Thomsen. *OLAP Solutions: Building Multidimensional Information Systems*. John Wiley & Sons, 1997.

[TSC99] Erik Thomsen, George Spofford, and Dick Chase. *Microsoft OLAP Solutions*. John Wiley & Sons, Inc., 1999.

[W3C99] W3C. XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`, November 1999. Current as of June 15, 2001.

[W3C00] W3C. Extensible Markup Language (XML) 1.0 (Second Edition). `http://www.w3.org/TR/REC-xml`, October 2000. Current as of June 15, 2001.

[W3C01a] W3C. XML Schema Part 0: Primer. `http://www.w3.org/TR/xmlschema-0`, May 2001. Current as of June 15, 2001.

[W3C01b] W3C. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`, June 2001. Current as of June 15, 2001.

[ZL96] Qiang Zhu and Per-Åke Larson. Global Query Processing and Optimization in the CORDS Multidatabase System. In *Proceedings of the 9th ISCA International Conference on Parallel and Distributed Computing Systems*, pp. 640–646, 1996.

A Syntax of SQL$_M$ and SQL$_{XM}$

The considered subset of SQL is given by the following syntax:

```
<query>                ::=  SELECT <select list>
                            FROM Cube | <query>
                            [WHERE <where predicate>]
                            [GROUP BY <group by list>
                            [HAVING <having predicate>]]
<select list>          ::=  <select name>
                        |   <select list>, <select name>
<select name>          ::=  <level name>
                        |   <aggregate function>(Measure)
<aggregate function>   ::=  DEFAULT | COUNT | SUM | MIN | MAX | AVG
<where predicate>      ::=  NOT(<where predicate>)
                        |   <where predicate> AND|OR <where predicate>
                        |   (<where predicate>)
                        |   <where expression>
<where expression>     ::=  <where name> <predicate operator> <where name>
                        |   <where name> <predicate operator> <value>
                        |   <where name> IN (<value list>)
                        |   <where name> LIKE ‘String’
<where name>           ::=  <level name>
                        |   Measure
<group by list>        ::=  <group by name>
                        |   <group by name>, <group by list>
<group by name>        ::=  <level name>
<having predicate>     ::=  NOT(<having predicate>)
                        |   <having predicate> AND|OR <having predicate>
                        |   (<having predicate>)
                        |   <having expression>
<having expression>    ::=  <having name> <predicate operator> <having name>
                        |   <having name> <predicate operator> <value>
                        |   <having name> IN (<value list>)
                        |   <having name> LIKE ‘String’
<having name>          ::=  <select name>
<level name>           ::=  Level
                        |   Level(Level)
<predicate operator>   ::=  <> | = | > | >= | < | <=
<value>                ::=  ‘String’ | Real | Integer
<value list>           ::=  <value>
                        |   <value>, <value list>
```

Allowing level expressions in the SELECT, WHERE, GROUP BY, and HAVING clauses causes the following changes to this syntax:

```
<select name>            ::=  <level name>
                          |   <aggregate function>(Measure)
                          |   <decoration expression>
<where name>             ::=  <level name>
                          |   Measure
                          |   <selection expression>
<having name>            ::=  <level name>
                          |   <aggregate function>(Measure)
                          |   <selection expression>
<group by name>          ::=  <level name>
                          |   Measure
                          |   <decoration expression>
<decoration expression>  ::=  Level [ ’[’ <semantic modifier> ’]’ ] [/Link]/ <xpath expression> [AS Level]
```

<selection expression> ::= Level [/Link]/ <xpath expression>

B Inlining XML Data in Predicates

This section describes how to transform a predicate containing level expressions to a new predicate without the level expressions but instead containing references to the XML data. This is done by defining a transformation function that, given a predicate with level expressions and the data that is referred to in the predicate, returns an equivalent predicate without the level expressions. We consider five different uses of level expressions in the predicates: comparing level expressions to a constant, a level, a measure, and a sequence of constants using the IN operator, and finally comparing XML data returned from two different level expressions.

Definition B.1 The transformation function $\mathcal{T} : FederationPredicates \mapsto CubePredicates$ is defined recursively as follows.

$$\mathcal{T}(p) = \begin{cases} \mathcal{T}(p_1)\ bo\ \mathcal{T}(p_2) & \text{if } p = p_1\ bo\ p_2, \\ NOT(\mathcal{T}(p_1)) & \text{if } p = NOT(p_1), \\ p_{const} & \text{if } p = L/Link/xp\ po\ K, \text{ where } K \text{ is a constant value}, \\ p_{level} & \text{if } p = L_1/Link/xp\ po\ L_2, \text{ where } L_2 \text{ is a level}, \\ p_{measure} & \text{if } p = L/Link/xp\ po\ M, \text{ where } M \text{ is a measure}, \\ p_{levexp} & \text{if } p = L_1/Link_1/xp_1\ po\ L_2/Link_2/xp_2, \\ p_{in} & \text{if } p = L/Link/xp \text{ IN } (K_1, \ldots, K_n), \text{ where } K_i \text{ is constant value}, \\ p & \text{Otherwise}. \end{cases}$$

where the binary operator bo is either AND or OR, and the predicate operator po is one of: $=, <, >, <>, >=, <=$, and LIKE.

The new predicates are defined in the following. □

Since the function is only applied recursively to parts of a predicate that are smaller than the original predicate the recursion will always terminate.

For each of the predicate types defined below the approximate length of the resulting predicate is given.

p_{const}**:** The idea is to construct a set of dimension values being linked to nodes that satisfy the predicate and then require the values in the level to be equal to one of these values:

$\mathcal{T}(L/Link/xp\ po\ K) =$ "L IN $(t_1, \ldots, t_n)$"

where $t_i \in \{t | \forall(e, s) \in Link(t \in \text{StrVal}(s_j) \wedge s_j \in xp(s) \wedge s_j\ po\ K = true)\}$. That is, a string t_i is added to the set if the node it represents satisfies the predicate.

The length of the predicate is $Length_{const} = O(n)$.

p_{level}**:** A level L_2 is be compared to a set of nodes resulting from a level expression $L_1/Link/xp$. This comparison is done by comparing the level L_2 to each of the nodes in the set and if just one of these new predicates is satisfied for some value e_2 in L_2 the tuple containing e_2 should be included in the result. Note that this is only true for tuples containing the value in L_1 that generated the set of nodes. Thus, L_2 must be compared to all the node sets generated by values in L_1:

$$\begin{aligned} \mathcal{T}(L_1/Link/xp\ po\ L_2) = \text{"}&((L_2\ po\ t_{11} \text{ OR } \ldots \text{ OR } L_2\ po\ t_{1k_1}) \text{ AND } L_1 = e_1) \\ &\text{OR } \ldots \text{ OR} \\ &((L_2\ po\ t_{n1} \text{ OR } \ldots \text{ OR } L_2\ po\ t_{nk_n}) \text{ AND } L_1 = e_n)\text{"} \end{aligned}$$

for all $e_i, \ldots, e_n$ and $t_{11}, \ldots, t_{nk_n}$ where $\forall(e_i, s) \in Link(e_i \in L_1 \wedge t_{ij} \in \text{StrVal}(s_j) \wedge s_j \in xp(s))$.

Intuitively we only make a sequence of disjunctions for the values in L_1 that participate in the link, since only these values generate a set of nodes. Values that don't participate in the link should never be included in the result because we use *any* semantics.

The length of the predicate is $Length_{level} = O(\sum_{i=1}^{n} k_i)$.

$p_{measure}$**:** Comparisons between measures and level expressions are treated similarly to p_{level}.

p_{levexp}**:** This is the result of comparing two level expressions. This means that we must compare two sets of nodes to determine whether or not to include a tuple in the result: one node set generated by the level expression $L_1/link_1/xp_1$ and one generated by $L_1/link_2/xp_2$. Intuitively a tuple containing $e_1 \in L_1$ and $e_2 \in L_2$ should be included if the two node sets generated by e_1 and e_2 have some common node. In other words it should be included if the intersection of the two sets is non-empty.

For ease of presentation a disjunctive clause on the form $(t_{ab}\ po\ r_{cd}$ OR ... OR $t_{ab}\ po\ r_{ce})$ is abbreviated as $P_{ab,cd,ab,ce}$.

$$\begin{aligned}
&\mathcal{T}(L_1/link_1/xp_1\ po\ L_1/link_2/xp_2) = \\
&\text{“}(((P_{11,11,11,1l_1} \text{ OR} \ldots \text{OR } P_{1k_1,11,1k_1,1l_1}) \text{ AND } L_1 = e_{11} \text{ AND } L_2 = e_{21}) \text{ OR} \ldots \text{OR} \\
&\ ((P_{11,m1,11,ml_m} \text{ OR} \ldots \text{OR } P_{1k_1,m1,1k_1,ml_m}) \text{ AND } L_1 = e_{11} \text{ AND } L_2 = e_{2m})) \text{ OR} \ldots \text{OR} \\
&\ (((P_{n1,11,n1,1l_1} \text{ OR} \ldots \text{OR } P_{nk_n,11,nk_n,1l_1}) \text{ AND } L_1 = e_{1n} \text{ AND } L_2 = e_{21}) \text{ OR} \ldots \text{OR} \\
&\ ((P_{n1,m1,n1,ml_m} \text{ OR} \ldots \text{OR } P_{nk_n,m1,nk_n,ml_m}) \text{ AND } L_1 = e_{1n} \text{ AND } L_2 = e_{2m}))\text{”}
\end{aligned}$$

where $\forall(e_{1i}, s_1) \in Link(e_{1i} \in L_1 \wedge t_{ij} \in \text{StrVal}(s_j) \wedge s_j \in xp(s_1))$
and $\forall(e_{2i}, s_2) \in Link(e_{2i} \in L_2 \wedge r_{ij} \in \text{StrVal}(s_j) \wedge s_j \in xp(s_2))$
for all $e_{1i}, \ldots, e_{1n}$, $t_{11}, \ldots, t_{nk_n}$, $e_{2i}, \ldots, e_{2m}$, and $r_{11}, \ldots, r_{mk_m}$.

The length of the predicate is $Length_{levexp} = O((\sum_{i=1}^{n} k_i) \cdot (\sum_{i=1}^{m} l_i))$.

p_{in}**:** The IN operator is a shorthand for a series of equalities involving a constant. Thus, the last predicate p_{in} is simply constructed from the new predicate for constant expressions p_{const}:

$$\mathcal{T}(L/Link/xp \text{ IN } (K_1, \ldots, K_n)) = \mathcal{T}(L/Link/xp = K_1) \text{ OR } \ldots \text{ OR } \mathcal{T}(L/Link/xp = K_n))$$

Thus, the length of the predicate is $Length_{in} = O(\sum_{i=0}^{n} Length_{level,i})$.