XML-Extended OLAP Querying

Dennis Pedersen Karsten Riis Torben Bach Pedersen

Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark
{dennisp,riis,tbp}@cs.auc.dk

Abstract

The rapidly changing data requirements of today's dynamic business environments are not handled well by current On-Line Analytical Processing (OLAP) systems. Physically integrating data from new sources into OLAP 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 external data will most often be available in XML format.

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 presented along with dimensional data in OLAP query results and enables the use of external XML data for selection and grouping. Special care is taken to ensure that semantic problems do not occur in the integration process. This opens up many new application areas for OLAP, e.g., in the B2B and scientific domains. A number of effective optimization techniques for OLAP-XML federations are presented. Performance results from the prototype implementation show that the approach is an attractive alternative to physical integration. The approach is exemplified using a real-world case study from the B2B domain.

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 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*. The multidimensional approach offers a number of advantages over traditional types of DBMSs, including automatic aggregation, visual querying, and good query performance due to the use of pre-aggregation [18]. However, it is difficult for OLAP systems to handle changing and unanticipated data requirements as physically integrating data can be a complex and time-consuming process requiring the cube to be rebuilt [18]. 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 of data is desirable, i.e., a *federated* database system [17] is called for. The increasing use of Extended Markup Language (XML), e.g. in B2B applications, suggests that the required external data will mostly 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 sometimes irregular structure of XML data means that problems related to correct aggregation of data can occur.

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. Second, external XML data may be used for selection. Third, OLAP data may be grouped by external XML data when aggregation is performed. Special care is taken to ensure that the possibly irregular structure of the XML data does not cause problems w.r.t. correct aggregation of data. A flexible linking mechanism is devised to associate cube data with parts of XML documents. We make no assumptions about the existence of Document Type Definitions (DTDs) or XML Schemas [20]. To demonstrate the capabilities of the approach, we present a data model and a multi-schema query language, *XML-Extended Multidimensional SQL* (SQL_{XM}), based on SQL and XPath [19]. SQL

and XPath are chosen for their simplicity, wide-spread use, and compact syntax. We also present a number of effective rule-based and cost-based optimization techniques for the approach, including algebraic query rewriting, inlining, optimizations over limited query interfaces, and caching and prefetching. A prototype implementation and experiments that show the performance of the approach and the effectiveness of the optimizations are also presented. The experiments suggest that the approach is in many cases an attractive alternative to physical integration.

As almost all data sources can be efficiently wrapped in XML format [1], the approach also allows external data from sources such as relational, object-relational, and object databases to be used in a powerful and flexible way, opening up new application areas for OLAP as data need no longer be integrated physically in the OLAP DB.

There has been a great deal of previous work on data integration, e.g., on integrating relational data [10], object-oriented data [16], semi-structured data [4], and a combination of relational and semi-structured data [6]. However, none of these handle the advanced issues related to OLAP systems, e.g., dimensions with hierarchies and the problems related to correct aggregation. This is also true for the combined relational and XML query language xQuery [21], and for nD-SQL [5], which considers the federation of relational sources providing basic OLAP functionality. One previous paper [15] 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 the external data and offer a more general use of external data when performing decoration, selection, and grouping. Query processing and optimization has been considered for data warehousing/OLAP systems [18], federated, distributed, and multi-databases [17], heterogeneous databases [2, 9], and XML and semistructured data [4]. However, previous work does not address the special case of optimizing OLAP queries in a federated environment.

We believe this paper to be the first to consider the integration of OLAP and XML data, including advanced issues such as dimension hierarchies and correct aggregation of data. Also, we belive to be the first to consider query processing and optimization for this setting.

The rest of the paper is organized as follows. Section 2 motivates our approach and presents the case study used throughout the paper. Section 3 defines the data models and query languages used in the federation components. Section 4 defines the linking mechanism and its use in OLAP-XML federations. Section 5 defines the semantics of the approach. Sections 6 and 7 describes the optimizations, and implementation and experiments, respectively. Section 8 concludes the paper and points to future work.

2 Motivation

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. Although our examples mostly come from the business world, scientific applications can also benefit heavily from this type of system. Indeed, in many scientific domains, there are already a number of data sources, e.g., the SWISSPROT protein databank, which are primarily accessed over the Internet. We believe that such data sources will publish their data in XML-based formats in the future. Also, statistical database users such as census agencies also have a long tradition of using information published on the internet, e.g., demographic information, in their analyses.

This federated approach where *users* are responsible for defining the federation, has been referred to as a *loosely coupled federation* [17]. 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 be disallowed because of copyright rules, or it may not be practical, e.g. because data changes too frequently. 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.

The federated approach also allows components to maintain the *high degree of autonomy* which is essential when data is accessed from sources outside the organisation controlling the federation, e.g., 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.

Case study The case study concerns the trading of electronic components. It is inspired by the Electronic Component Information Exchange (ECIX)[3], 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.

Customers use an OLAP database 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 1. 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. 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.

```
<?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 1. The Components Document

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.

3 Component Models

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 used, and an OLAP-extended version of SQL is used as the query language. The OLAP data model captures complex multidimensional data, e.g., irregular dimen-

sion hierarchies. We present just an overview of the model, mainly through the use of examples, the formal definition can be found in another paper [12]. For the XML component, the XPath data model and query language [19] is used, mainly because of its simplicity and wide-spread use.

The OLAP Data Model and Query Language 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). The first poset represent hierarchies of the *levels* which specify the possible levels of detail of the data. Each level is associated with a set of *dimension values*. The second poset represent the ordering of the dimension values, i.e., which values roll up to one another. Dimensions are used to capture the possible ways of grouping data. A *fact table* F is a relation containing one attribute for each dimension, and one attribute for each measure. Thus, An *n-dimensional cube* is a three-tuple consisting of a cube name, a non-empty set of dimensions, and a fact table. The *cube name* describes the type of facts contained in the cube.

Example 3.1 In the case study, we have a Time dimension, an ECs dimension and a Suppliers dimension. The Suppliers dimension consists of the levels Supplier, Country, and $\top_{Sup}$, which denotes *all* of the Supplier dimension. The ordering of the levels and the dimension values can be seen in Figure 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. From these parts, we can construct a three-dimensional cube with the cube name $Purchases$, the dimensions, levels, and ordering of dimension values as depicted in Figure 2, and the fact table from Table 1. "FF" and "L" are names of classes denoting "Flip-flops" and "Latches", respectively. □

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. Fact Table For Purchases Database

Next, we discuss the notion of *summarizability* and discuss how it is used to ensure *correct aggregation* when "rolling up" data from lower to higher levels of granularity. Summarizability is an important cube property as it states when lower-level aggregates, which are often precomputed, can be used to calculate higher-level aggregates, and when they 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 this paper's setting than in normal OLAP systems, as the irregular structure of XML data often will violate the summarizability property. 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* [11]. 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 often 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 2 is strict, onto, and covering (note that strictness is a property of the data, not the schema and that the two parents of, e.g., EC1234, are from different levels, meaning that the EC dimension is strict). We keep track of what data can be aggregated by using so-called *aggregation types* [12].

A formal algebra has been defined over the OLAP data model presented above. Two operators are defined: a selection operator $\sigma_{Cube}[p]$ for selecting only the desired facts based on the predicate p, and a generalized projection operator, Π_{Cube} for aggregating fact data to the desired level of detail, possibly projecting out un-desired dimensions. The formal definitions can be found in [12].

As the formal algebra is not suitable for end users, we have also defined a SQL-like query language in terms of the algebra. We use a slight extension of a subset of SQL, called "Multidimensional SQL" (abbreviated SQL_M), to query multidimensional cubes. SQL is chosen as the base language for its simplicity and wide-spread use. We illustrate the considered syntax with examples. The complete specification is given in [12].

Example 3.2 Calculate costs by class and supplier for suppliers located in UK where 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
```

□

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. The XML data model underlying the XPath language views an XML document as

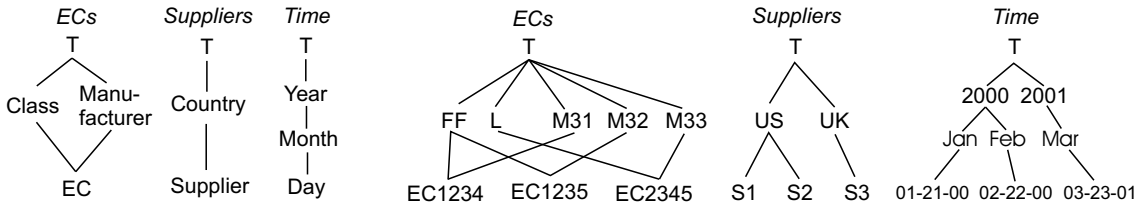

Figure 2. Schema (left) and instance (right) of the Purchases database.

a tree. Each node in the tree has one of the types: root, element, namespace, text, processing instruction, attribute, or comment. For more details about the XML data model and its use in our approach we refer to the XPath specification [19] and the full paper [14].

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$_n$. The returned set of nodes can also be restricted by applying one or more *predicates* which supports the usual boolean, mathematical, and string operators.

Example 3.3 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/Supplier/Class/Component[Manufacturer/MName = 'Johnson Components' OR Manufacturer/@MCode = 'M33'][../ClassName='Flip-flop']. The "../ClassName" notation finds an element named "ClassName" at any level in the document. □

For our purpose, we can abstract an XPath expression to be a function over a set of nodes:

Definition 3.1 (XPath Expression) Let S be a set of nodes in an XML document. An XPath expression is a function XP : $S \mapsto \mathcal{P}(S)$ 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

We now describe how *links* between an OLAP DB and external XML data can be used to make it easy for users to reference XML data in OLAP queries. The mechanism provides location transparency, since links can be changed without affecting existing queries. The fundamental linking mechanism is a relation between one dimension value in a cube and one node 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 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 document $x \in X$, and $Links$ is a set of links. The resulting link relation is given by: EnLink$(\{(e_1, x_1, locator_{x1}), \ldots, (e_k, x_k, locator_{xk})\}) = \{(e_i, s) | e_i \in \{e_1, \ldots, e_k\} \wedge s \in locator_{xi}(root(x_i))\}$, where e_i is a dimension value, x_i is an XML document, $locator_{xi}$ 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 must 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 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, e.g., UK, 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: 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} 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 of the form $L/link/xp$, 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 $\text{DefaultLink} : L \mapsto Links$, 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 $\text{DefaultLink}(EC)$ 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 collection of the cube in Example 3.1, the Components document, and the two links Sup_Link and EC_Link is a federation. We will refer to this as the Purchases federation in the following. □

5 Querying Federations

We now present an algebra over federations. 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.

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 [7]. 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 we consider how to use level expressions in the GROUP BY clause.

Example 5.1 Let "AllTimePurchases" be the aggregation of the Purchases cube to the EC and Supplier levels. The fact table of this cube is shown in Table 3(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  SUM(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] resulting in non-strictness in the new "decoration" dimension. 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 3(b), where a fact is created for each different description node resulting from the level expression. Another possibility is the result shown in Table 3(c), where an arbitrary node is picked and at most one fact is created for each EC. A third possibility is shown in Table 3(d), where all description nodes are concatenated. In all cases we use a special "N/A" (Not Available) value to indicate that no description is found for an EC. Note that EC1234 gets several description decoration values as the decoration expression EC/Description is only dependent on the EC dimension. □

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 3. Fact table And Decorations

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.

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. 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.

The user specifies the semantics of a decoration by giving a ANY, CONCAT, or ALL semantic modifier in the level expression, e.g., 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. The query in Example 5.1 actually results in Table 3(c) since ANY is the default.

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, create a new level in the same dimension as the starting level, or to keep the decorated data in an external component [15]. 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. Also, aggregation is still possible in the original dimensions, as these are not changed by decoration.

Intuitively, only two things are changed when decorating a cube: A new dimension is added and the fact table is updated to reflect this. 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.

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 5.3 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 (logically) 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 5.4 The above query is evaluated by first decorating the Purchases cube resulting in the fact table shown in Table 3(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. Consequently, if non-strictness occurs because of the decoration and if aggregation results in duplicate facts, this is handled by setting the aggregation type to c, preventing further aggregation. Formally, the generalized projection operator over federations is defined as follows:

Definition 5.1 (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: $X' = \{x \in X | \exists link \in Links'(\exists(e,s) \in link(s \in \text{Nodes}(x)))\}$. □

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 5.5 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 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 (logically) 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 5.6 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 two 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 5.2 (Selection over federations) Let $\mathcal{F} = (C, Links, X)$ be a federation, where C has dimensions $D_1, \ldots, D_n$ and measures $M_1, \ldots, M_l$. 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.

The formal semantics of a SQL_{XM} query is given in the full paper [14].

An SQL_{XM} query over a federation is (logically) 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 also require a decoration. Notice that the new decoration dimensions used for selection are not mentioned in the following generalized projection and are therefore removed after use. Since the new dimensions are never aggregated up to the decoration level, no changes are made to the aggregation types.

6 Query Optimization

A naive query processing strategy will process SQL_{XM} queries in three major steps. First, any XML data referenced

in the query is fetched and stored in a temporary database as relational tables. Second, a pure OLAP query is constructed from the SQL_{XM} query and evaluated on the OLAP data, resulting in a new table in the temporary database. Finally, these temporary tables are joined, and the XML-specific part of the SQL_{XM} query is evaluated on the resulting table. This strategy will only perform satisfactorily for rather small databases. The primary problems are that decoration operations require large parts of the OLAP and XML data to be transferred to temporary storage before decoration can take place, i.e., the primary bottleneck in the federation will most often be the moving of data from OLAP and XML components. Thus, our optimization efforts have focused on this issue. These efforts include both *rule based* and *cost based* optimization techniques.

The *rule based* optimization uses the heuristic of pushing as much of the query evaluation towards the components as possible. Although not always valid for more general database systems, this heuristic is always valid in our case since the considered operations all reduce the size of the result. The rule based optimization algorithm *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 aggregate data 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. The partitioning is based on *transformation rules* for the federation algebra. These include a set of novel rules for moving the decoration operator around the query tree. The rules not involving decoration are close to the rules for Multi-Set Extended Relational Algebra [8].

Three different *cost based* optimization techniques are employed. The use of cost based optimization requires a complete cost model and the estimation of several cost parameters which is described in another paper [13].

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, much more data than the result needs to be transferred to the temporary component. The proposed solution to this problem is to *inline* literal XML data values into OLAP predicates, i.e., to have the predicates refer to a list of literal constants rather than a XPath expression. 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. In summary, 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. For each of these three strategies, the total evaluation cost is estimated and the cheapest one is used.

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 is an efficient way to find a useful cached result for a given OLAP query. *Pre-fetching* is also employed for speeding up queries that have not been posed before. In summary, we perform caching and prefetching 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 stored to avoid constructing the same tables again. Currently, we do not cache or pre-fetch entire federation queries as the cache space they take up is in most cases too high in comparison with the probability that they can be re-used.

7 Implementation and Experiments

The overall architecture of the prototype is shown in Figure 4. 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 *source 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. The *meta data component* contains descriptions of the dimensions in the OLAP component, whereas the *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. In the prototype, the OLAP component is based on Microsoft Analysis Services and queried with the MDX language. The XML component is based on Software AG's Tamino XML Database system, which provides an XPath

interface. A single Oracle 8*i* system is used for all three auxiliary components.

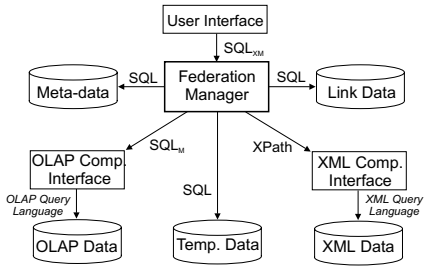

Figure 4. Prototype Architecture

We have performed a set of experiments to evaluate the performance of our federation approach and the effectiveness of the optimization techniques. Here, we only summarize the results, the details can be found in the full paper [14]. The experiments were performed on a 900 Mhz Pentium machine with 512MB of RAM and 20GB of disk. The cube is based on about 50 MB of data generated using the TPC-H benchmark 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. 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. Two natural links, NLink and Tlink, are defined to from Nation level in the cube to the Nation elements in the XML data, and from the Type level in the cube to the Type elements in the XML data, respectively.

For pure decoration queries, we found that the overhead of performing decoration was low 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.

For grouping and selection queries, we found that the overhead caused 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. For selections, the OLAP result is often comparable in size to the final result for queries where the predicate refers to decorations of levels that must be present in the result. This is true unless the predicate is very selective.

The experiments with the inlining technique compared three alternatives: no inlining, a simple inlining strategy, and cost-based inlining. Where no inlining is used, there is a very large overhead as the OLAP query can only aggregate to lower levels. As a consequence, not only the OLAP query but also the temporary component query take much longer to evaluate. The use of the simple inlining strategy, "Always use inlining if the predicate is simple" is significantly faster because the OLAP query can now aggregate to higher levels. 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 higher level contains a large number of dimension values. Also, the OLAP query cannot be evaluated until all XML data has been retrieved. Consequently, it sometimes faster to inline only some predicate data, i.e., as determined by the cost based inlining strategy. This required only one OLAP query. 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. In summary, the simple inlining strategy improved performance by a factor of 4.5 over no inlining, while the cost-based strategy was 3 times faster than simple inlining and 14 times faster than no inlining.

The experiments with caching/prefetching showed a performance improvement by factors of 4–25 over no caching or prefetching, depending on the query type, with decoration and grouping queries gaining the most improvement.

We have also performed experiments that compare our approach to the performance-wise ideal situation where the external data is *physically integrated* in the OLAP cube. These experiments were performed with 1GB of TPC-H data in the OLAP cube, plus 100MB used for pre-computed aggregates. In the XML component, we had 10 MB of XML data. The results of these experiments are seen in Figure 5. The "O" bars show the time spent in the OLAP component, while the "T" bars show the time spent in the temporary component. We compared three typical queries that performed decoration (1), selection (2), and grouping (3) using XML data, respectively. The experiments show that for typical queries the overhead of our approach was only 25–50% compared to physical integration. To conclude, the optimizations discussed above suggests that an SQL_{XM} query can in most cases be evaluated with a level of efficiency comparable to that of physical integration, while avoiding the problems related to physical integration in dynamic environments.

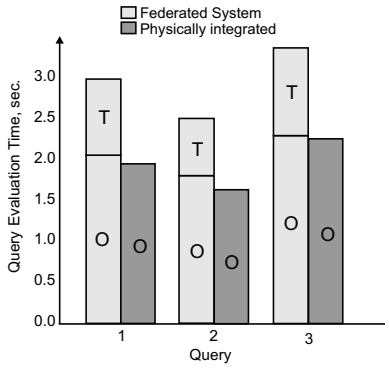

Figure 5. Performance Comparison

8 Conclusion and Future Work

Many OLAP systems operate in highly dynamic environments where changes in data requirements are common and data changes frequently, meaning that logical OLAP federations are far more feasible than physically integrated databases. As external data will most often be available in XML format, it should be possible to integrate OLAP and XML data seamlessly.

In this paper we presented an approach for the federation of OLAP and XML data, given in terms of a formal data model and algebraic query language. To demonstrate our approach we introduced a federated query language, SQL_{XM}, incorporating the XML query language XPath into a subset of SQL adapted to multidimensional querying. SQL_{XM} allows XML data to be used directly in an OLAP query to *decorate* multidimensional cubes with external XML data, and to *group* and *select* cube data based on XML data values. The incorporation of XML data in cubes was made such that semantic problems were avoided, e.g., when aggregation was performed on the resulting cube no double-counting of data could occur. A number of effective optimization techniques for OLAP-XML federations were described. Finally, a prototype implementation and a set of experiments stating the performance of the approach and the effectiveness of the optimization techniques were described, showing the attractiveness of the approach compared to physical integration.

We believe this paper to be the first to consider the integration of OLAP and XML data, including advanced issues such as dimension hierarchies and correct aggregation of data. Also, we believe to be the first to consider query processing and optimization for this setting.

In future work, interesting research issues include how to capture the document order of an XML document in the result of an OLAP query, how to incorporate new XML-based measures into a cube, and how extra structural information about the XML data, such as DTDs and XML Schemas, can be utilized. Also, other query languages than SQL and XPath could be considered for the federation.

References

[1] V. Christophides, S. Cluet, J. Simeon. On Wrapping Query Languages and Efficient XML Integration. In *Proc. of SIGMOD*, pp. 141–152, 2000.

[2] W. Du, R. Krishnamurthy, and M.-C. Shan. Query Optimization in a Heterogeneous DBMS. In *Proceedings of VLDB*, pp. 277–291, 1992.

[3] ECIX Quickdata Architecture. `www.si2.org/ecix/`. Current as of June 15, 2001.

[4] Garcia-Molina H. et al. The TSIMMIS Approach to Mediation: Data Models and Languages. *JIIS* 8(2), pp. 117–132, 1997.

[5] F. Gingras and L. V. S. Lakshmanan. nD-SQL: A Multi-Dimensional Language For Interoperability and OLAP. In *Proc. of VLDB*, pp. 134–145, 1998.

[6] R. Goldman and J. Widom. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In *Proc. of SIGMOD*, pp. 285–2.596, 2000.

[7] J. Gray et al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Proc. of ICDE*, pp. 152–159, 1996.

[8] P. W. P. J. Grefen and R. A. de By. A Multi-Set Extended Relational Algebra - A Formal Approach To A Practical Issue. In *Proceedings of ICDE*, pp. 80–88, 1994.

[9] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proceedings of VLDB*, pp. 276–285, 1997.

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

[11] H-J. Lenz and A. Shoshani. Summarizability in OLAP and Statistical Databases. In *Proc. of SSDBM*, pp. 39–48, 1997.

[12] D. Pedersen, K. Riis, and T. B. Pedersen. A Powerful and SQL-Compatible Data Model and Query Language for OLAP. In *Proc. of ADC*, pp. 121–130, 2002.

[13] D. Pedersen, K. Riis, and T. B. Pedersen. Cost Modeling and Estimation for OLAP-XML Federations. In *Proc. of DawaK*, 10 pages, to appear, 2002.

[14] D. Pedersen, K. Riis, and T. B. Pedersen. XML-Extended OLAP Querying. *Technical Report TR 02-5001, Department of Computer Science, Aalborg University*, 20 pages, 2002.

[15] T. B. Pedersen et al. Extending OLAP Querying to External Object Databases. In *Proc. of CIKM*, pp. 405–413, 2000.

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

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

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

[19] W3C. Xml path language (xpath) version 1.0. `www.w3.org/TR/xpath`. Current as of June 15, 2001.

[20] W3C. Xml schema part 0: Primer. `www.w3.org/TR/xmlschema-0/`. Current as of June 15, 2001.

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