Cost Modeling and Estimation For OLAP-XML Federations

Authors: Dennis Pedersen
Karsten Riis
Torben Bach Pedersen

Technical Report 02-5003
Department of Computer Science
Aalborg University

Created on August 14, 2002

Cost Modeling and Estimation For OLAP-XML Federations

Dennis Pedersen dennisp@cs.auc.dk
Karsten Riis riis@cs.auc.dk
Torben Bach Pedersen tbp@cs.auc.dk

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

Abstract

The ever-changing data requirements of today's dynamic businesses are not handled well by current On-Line Analytical Processing (OLAP) systems. Physical integration of unexpected data into OLAP systems is a long and time-consuming process, making logical integration, or *federation*, the better choice in many cases. The increasing use of XML, e.g. in business-to-business (B2B) applications, suggests that the required data will often be available in XML format. Thus, federations of OLAP and XML databases will be very attractive in many situations. However, a naive implementation of OLAP-XML federations will not perform well enough to be useful. In an efficient implementation, cost-based optimization is a must, creating a need for an effective cost model for OLAP-XML federations. However, existing cost models do not support such systems.

In this paper we present a *cost model* for OLAP-XML federations, along with techniques for *estimating* the cost model parameters in a federated OLAP-XML environment. The paper also present the cost models for the OLAP and XML components in the federation on which the federation cost model is built. The cost model has been used as the basis for effective cost-based query optimization in OLAP-XML federations. Experiments show that the cost model is precise enough to make a substantial difference in the query optimization process.

1 Introduction

OLAP systems [20] enable powerful decision support based on multidimensional analysis of large amounts of detail data. OLAP data are often organized in multidimensional *cubes* containing *measured values* that are characterized by a number of hierarchical *dimensions*.

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 [20]. Thus, logical, rather than physical, integration is desirable, i.e. a *federated* database system [17] is called for. The increasing use of Extended Markup Language (XML)[22], 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 access XML data from an OLAP system, i.e., OLAP-XML federations are needed. For the implementation of OLAP-XML federations to perform satisfactorily, cost-based optimization is needed. This, in turn, creates a need for an effective cost model for OLAP-XML federations. However, existing cost models do not support such systems.

In this paper we present such a cost model for OLAP-XML federations, along with techniques for estimating cost model parameters in an OLAP-XML federation. We also present the cost models for the autonomous OLAP and XML components in the federation on which the federation cost model is based. The cost model has been used as the basis for effective cost-based query optimization in OLAP-XML federations, and experiments show that the cost model is powerful enough to make a substantial difference in the query optimization process.

A great deal of previous work on data integration exist, e.g., on integrating relational data [7], object-oriented data [16], and semi-structured data [2] However, none of these handle the issues related to OLAP systems, e.g., dimensions with hierarchies. One paper [15] has considered federating OLAP and object data, but does not consider cost-based query optimization, let alone cost models. To our knowledge, no general models exist for estimating the cost of an OLAP query (MOLAP or ROLAP). Shukla et al. [18] describe how to estimate the size of a multidimensional aggregate, but their focus is on estimating storage requirements rather than on the cost of OLAP operations. Also, we know of no cost models for XPath queries, the closest related work being on path expressions in object databases [11]. Detailed cost models have been investigated before for XML [9], relational components [4], federated and multidatabase systems [5, 7, 16, 25, 26], and heterogeneous systems [3, 10]. However, since our focus is on OLAP and XML data sources only, we can make many assumptions that permit better estimates to be made. Several techniques have been used in the past for acquiring cost information from federation components. A commonly used technique is *query probing* [24] where special queries, called *probing queries*, are used to determine cost parameters. *Adaptive cost estimation* [8] is used to enhance the quality of cost information based on the actual evaluation costs of user queries.

We believe this paper to be the first to propose a cost model for OLAP-XML federation queries, along with techniques for estimating and maintaining the necessary statistics. Also, the proposed cost models for the autonomous OLAP and XML components are believed to be novel.

The rest of the paper is organized as follows. Section 2 describes the basic concepts of OLAP-XML federations. Section 3 describes the federation system architecture and outlines the query optimization strategy. Sections 4, 4, 5, and 6 presents the federation cost model and the cost models for the OLAP and XML components. Section 8 presents a performance study evaluating the cost model. Section 9 summarizes the paper and points to future work. Appendices A, B, and C contain additional detail on the OLAP component cost model, the XML component cost model, and the queries performed in the experiments, respectively.

2 OLAP-XML Federation Concepts

This section briefly describes the concepts underlying the OLAP-XML federations that the cost model is aimed at. These concepts are described in detail in another paper [13]. The examples in the paper is based on a case study concerning B2B portals, where a cube tracks the cost and number of units for *purchases* made by customer companies. The cube has three dimensions: Electronic Component (EC), Time, and Supplier. External data is found in an XML document that tracks component, unit, and manufacturer information. The details of the case study are also described in [13].

The OLAP data model is defined in terms of a multidimensional *cube* consisting of a *cube name*, *dimensions*, and a *fact table*. Each dimension has a hierarchy of the *levels* which specify the possible levels of detail of the data. Each level is associated with a set of *dimension values*. Each dimension also captures the hierarchy of the dimension values, i.e., which values roll up to one another. Dimensions are used to capture the possible ways of grouping data. The actual data that we want to analyze is stored in a *fact table*. A fact table is a relation containing one attribute for each dimension, and one attribute for each *measure*, which are the properties we want to aggregate, e.g., the sales price. The cubes can contain *irregular* dimension hierarchies [12] where the hierarchies are not balanced trees. The data model captures this aspect as it can affect the *summarizability* of the data [12], i.e., whether aggregate computations can be performed without problems. The data model is equipped with a formal algebra, with a *selection operator* for selecting fact data, σ_{Cube}, and a *generalized projection operator*, Π_{Cube}, for aggregating fact data. On top of the algebra, a SQL-based OLAP query language, SQL_M, has been defined. For example, the SQL_M query "SELECT SUM(Quantity),Nation(Supplier) FROM Purchases GROUP BY Nation(Supplier)" computes the total quantity of the purchases in the cube, grouped by the Nation level of the Supplier dimension. The OLAP data model, algebra, and the SQL_M query language is described in detail in another paper [12].

Extended Markup Language (XML) [22] specifies how documents can be structured using so-called *elements* that contains *attributes* with atomic values. Elements can be nested within and contain references to each

other. For example, for the example described above, a “Manufacturer” element in the XML document contains the “MCode” attribute, and is nested within a “Component.” XPath [22] is a simple, but powerful language for navigating within XML documents. For example, the XPath expression “Manufacturer/@Mcode” selects the “MCode” attribute within the “Manufacturer” element.

The OLAP-XML federations are based on the concept of *links* which are relations linking dimension values in a cube to elements in an XML document, e.g., linking electronic components (ECs) in the cube to the relevant “Component” elements in the XML document. A *federation* consists of a cube, a collection of XML documents, and the links between the cube and the documents. The most fundamental operator in OLAP-XML federations is the *decoration* operator which basically attaches a new dimension to a cube based on values in linked XML elements. Based on this operator, we have defined an extension of SQL_M, called SQL_{XM}, which allows XPath queries to be added to SQL_M queries, allowing linked XML data to be used for decorating, selecting, and grouping fact data. For example, the SQL_{XM} query “SELECT SUM(Quantity),EC/Manufacturer/@MCode FROM Purchases GROUP BY EC/Manufacturer/@MCode“ computes total purchase quantities grouped by the manufacturer’s MCode which is found only in the XML document. The OLAP-XML federation concepts are described in detail in another paper [13].

3 The Federation System

In this section we give an overview of the OLAP-XML federation system, the design considerations and optimization techniques as well as their use in the federation system.

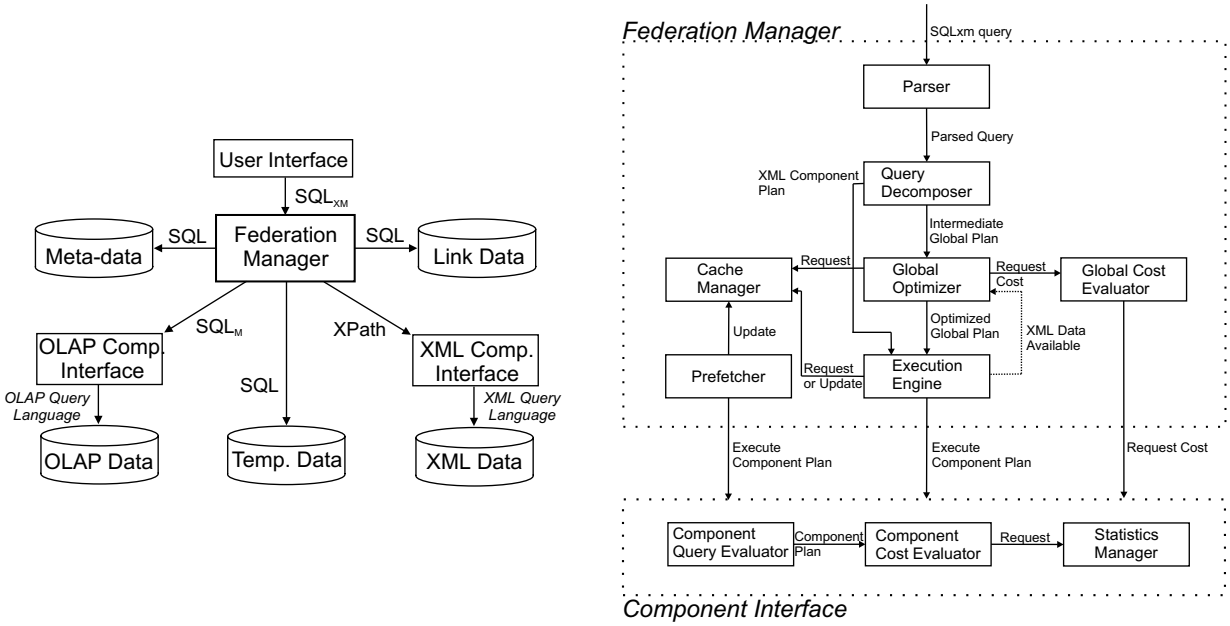

Figure 1: Architecture Of The Federation System and The Federation Manager

The overall architectural design of a prototype system supporting the SQL_{XM} query language is seen to the left in Figure 1. 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 irregular dimension 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. A partial prototype implementation has been performed to allow us to make performance experiments. In the prototype, the OLAP component uses Microsoft Analysis Services and is queried with MDX and SQL. The XML component is based on Software AG’s Tamino XML Database system [19], which

provides an XPath-like interface. For the external temporary component, a single Oracle 8*i* system is used.

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 optimization techniques are described in detail in another paper [14].

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 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 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. The most important *cost based* optimization 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 levels not present in the result, too much data needs to be transferred to the temporary component. Our solution to this problem is to *inline* XML data values into OLAP predicates as literals. However, this is not always a good idea 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 the alternatives.

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.

We now outline how the techniques discussed above are used in combination, referring to the components seen the right in Figure 1. 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. The cost estimates are 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.

4 Federation Cost Model

We now present a basic cost model for OLAP-XML federations. queries, and then outline the optimization techniques that, in turn, leads to a refined cost model.

Basic Federation Cost Model: 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 high-level 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 later. 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.

Refined Federation Cost Model: As discussed above, 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.

Level expressions can be inlined compactly into some types of predicates [14]. Even though it is always possible to make this inlining [14], 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 [14]. 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 4.1 Consider the predicate: "EC/Manufacturer/@MCode = Manufacturer(EC)". The decoration data for the level expression results in the following relationships between dimension 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 5.

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 4.2 In Figure 2, 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. □

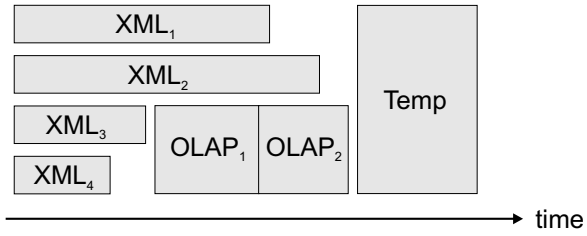

Figure 2: Total Evaluation Times for the Component Queries for A Single Federation Query

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 later, 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.

Component Cost Models: We now describe the models for the component costs that were used in the cost formulas in Section 4. This cost information is collected by the *Statistics Manager* and used by the cost evaluators seen to the right in Figure 1. The amount of cost information available to the federation may vary between federation components. Du et al. [3] 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 high degree of autonomy must be expected for XML components, while OLAP components (which are in-house) may or may not provide access to cost information. Thus, we assume 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.

5 OLAP Component Cost Model

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.

OLAP Component Statistics The estimation of cost parameters is based on statistical information represented by the functions shown in Table 1.

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 1: 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[4]. 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. [18] 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 [18] 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 [6] 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.

Estimating and maintaining OLAP cost parameters: The estimation and maintenance of the OLAP cost parameters is performed using *probing queries* as the OLAP component is assumed to be non-conforming. The initial estimations are gradually adjusted based on the actual evaluation times using an *adaptive* approach. Due to space constraints, we cannot give the details here, they can be found in Appendix A.

6 XML Component Cost Model

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, a relational or OO database or a specialized XML database [19]. 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 [9]. 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 [14]. Because XML components are non-conforming, the estimates are based on posing probing queries to the XML component to retrieve the necessary statistics.

XML Component Statistics: 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 described 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 [1], 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.

Obtaining and Using XML Component Statistics: Some of the information discussed above can be obtained directly if an XML Schema is available [23]. In that case, information such as *NodeSize*, *Cardinality*, or *Fanout* is determined from the schema. DTDs [22] 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. Due to space constraints, we cannot give the details here, they can be found in Appendix B.

7 Temporary Component Cost Model

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 [3]. This model has been demonstrated to provide good results for different DMBSs.

8 Experimental Results

The primary purpose of any cost model is to be used in the query optimization process. Thus, the most relevant quality measure of a cost model is whether it can guide the query optimizer to perform the right choices, rather

than how closely the model can predict the actual execution times. Following this reasoning, the experiments show the effect of using the cost model in the optimization process.

The experiments are based on about 50 MB of data generated using the TPC-H benchmark [21] 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. The queries that were used in the experiments are found in Appendix C. The results are shown in Figure 3. 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.

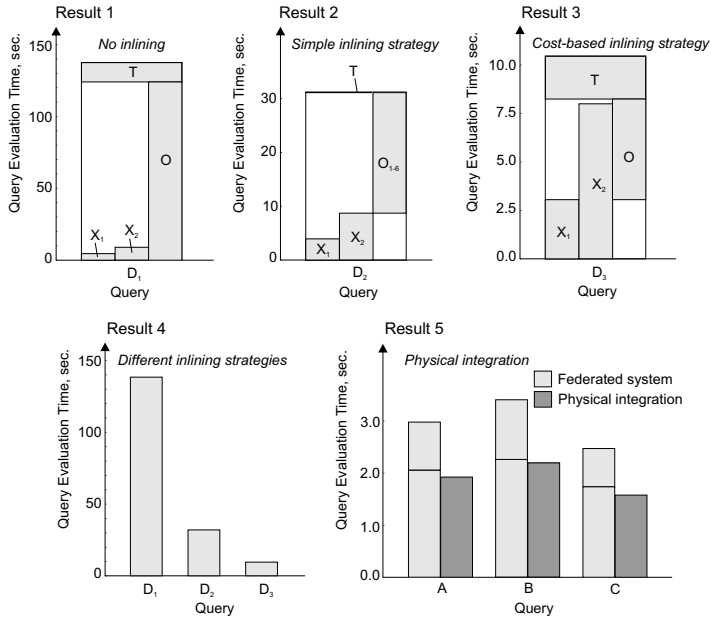

Figure 3: Experimental results. Notice that different time intervals are used in the graphs.

As described in Section 3, the cost model is not needed in the process of partitioning the queries, e.g., moving operations up and down the query tree, as the rule-based approach will always find the best solution. Thus, we only need to illustrate the use of the cost model in the cost-based part of the optimization process, namely the choice of what XML data to *inline* in the OLAP queries. The result of this is shown in Result 1–4. Result 1 shows the query execution time with no inlining. In Result 2, we use the simple heuristic "always use inlining if the predicate is simple," which cause inlining to be used for both predicates in the query. Finally, in Result 3, we use the cost model to determine where to use inlining (only one predicate is inlined). As the time intervals in Result 1–3 are different, Result 4 provides an overview of the results. It is seen that simple inlining is around 5 times faster no inlining, but that cost-based inlining is 3 times faster than simple inlining and an amazing 15 times faster than no inlining. This result shows that the cost model is powerful enough to make a substantial difference in the query optimization process. Result 8 compares these results to the highly optimized situation where all data is stored in the OLAP component, for a larger data set based on 1 GB of

TPC-H data, showing three different uses of XML data, namely in a decoration query (A), in a grouping query (B), and in a selection query (C). We see that the federated OLAP-XML approach is only 30% slower than the integrated OLAP approach (which is very inflexible w.r.t. changing data), a surprising result that is largely due to the effectiveness of the query optimization process and the cost model.

9 Conclusion

The ever-changing data requirements of today's dynamic businesses are not handled well by OLAP systems. Physical integration of new data into OLAP systems is a long and time-consuming process, making logical integration, or *federation*, the better choice in many cases. The increasing use of XML, e.g., in B2B applications, suggests that the required data will often be available in XML format. Thus, federations of OLAP and XML databases will be very attractive in many situations. This creates a need for an efficient implementation of OLAP-XML federations which requires cost-based optimization, and, in turn, an effective cost model for OLAP-XML federations.

Motivated by this need we have presented a cost model for OLAP-XML federations. Also, techniques for estimating cost model parameters in an OLAP-XML federation were presented. The paper also presented the cost models for the autonomous OLAP and XML components on which the federation cost model was based. The cost model was used as the basis for effective cost-based query optimization process in OLAP-XML federations. Experiments showed that the cost model is powerful enough to make a substantial difference in for the query optimization.

We believe this paper to be the first to propose a cost model for OLAP-XML federation queries, along with techniques for estimating and maintaining the necessary statistics for the cost models. Also, we believe the proposed cost models for the autonomous OLAP and XML components to be novel.

Future work will focus on how to provide better cost estimates when querying autonomous OLAP components and, in particular, autonomous XML components. For this purpose, extensive testing of commercial OLAP and XML database systems is needed. Also, better ways of obtaining cost parameters from the OLAP and XML components can probably be derived by an extensive examination of commercial systems.

References

[1] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the open oodb query optimizer. In *Proceedings of the SIGMOD Conference*, pp. 287–296, 1993.

[2] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of the 16th Meeting of the Information Processing Society of Japan*, pp. 7–18, 1994.

[3] W. Du, R. Krishnamurthy, and M.-C. Shan. Query Optimization in a Heterogeneous DBMS. In *Proceedings of the 18th VLDB Conference*, pp. 277–291, 1992.

[4] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, third edition, 2000.

[5] G. Gardarin, F. Sha, and Z.-H. Tang. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. In *Proceedings of the 22nd VLDB Conference*, pp. 378–389, 1996.

[6] G. Graefe, R. Bunker, and S. Cooper. Hash Joins and Hash Teams in Microsoft Sql Server. In *Proceedings of 24th VLDB Conference*, pp. 86–97, 1998.

[7] J. M. Hellerstein, M. Stonebraker, and R. Caccia. Independent, Open Enterprise Data Integration. *IEEE Data Engineering Bulletin*, 22(1):43–49, 1999.

[8] H. Lu, K.-L. Tan, and S. Dao. The Fittest Survives: An Adaptive Approach to Query Optimization. In *Proceedings of 21st VLDB Conference*, pp. 251–262, 1995.

[9] J. McHugh and J. Widom. Query Optimization For XML. In *Proceedings of 25th VLDB Conference*, pp. 315–326, 1999.

[10] H. Naacke, G. Gardarin, A. Tomasic. Leveraging Mediator Cost Models with Heterogeneous Data Sources. In *Proceedings of the 14th ICDE Conference*, pp. 351–360, 1998.

[11] C. Ozkan, A. Dogac, and M. Altinel. A Cost Model for Path Expressions in Object-Oriented Queries. *Journal of Database Management* 7(3), 1996.

[12] D. Pedersen, K. Riis, and T. B. Pedersen. A Powerful and SQL-Compatible Data Model and Query Language for OLAP. To appear in *Proceedings of the 13th ADC Conference*, 10 pages, 2002.

[13] D. Pedersen, K. Riis, and T. B. Pedersen. XML-Extended OLAP Querying. *Submitted for publication*, 2002.

[14] D. Pedersen, K. Riis, and T. B. Pedersen. Query Processing and Optimization for OLAP-XML Federations. *Submitted for publication*, 2002.

[15] T. B. Pedersen, A. Shoshani, J. Gu, and C. S. Jensen. Extending OLAP Querying To External Object Databases. In *Proceedings of the 9th CIKM Conference*, pp. 405–413, 2000.

[16] M. T. Roth, F. Ozcan, and L. M. Haas. Cost models do matter: Providing cost information for diverse data sources in a federated system. In *Proceedings of 25th VLDB Conference*, pp. 599–610, 1999.

[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] A. Shukla, P. Deshpande, J. F. Naughton, and K. Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. In *Proceedings of 22nd VLDB Conference*, pp. 522–531, 1996.

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

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

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

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

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

[24] Q. Zhu and P.-Å. Larson. Global Query Processing and Optimization in the CORDS Multidatabase System. In *Proceedings of the 9th PDCS Conference*, pp. 640–646, 1996.

[25] Q. Zhu and P.-Å. Larson. Developing Regression Cost Models for Multidatabase Systems. In *Proceedings of the 4th PDIS Conference*, pp. 220–231, 1996.

[26] Q. Zhu, Y. Sun, and S. Motheramgari. Developing Cost Models with Qualitative Variables for Dynamic Multidatabase Environments. In Proceedings of the 16th ICDE Conference, pp. 413–424, 2000.

A Details of the OLAP Component Cost Model

This section provides detail on the estimation and maintenance of the OLAP cost parameters.

Estimating OLAP Cost Parameters: The cost parameters $t_{OLAP,OH}$, $t_{OLAP,Eval}$, and $t_{OLAP,Trans}$ are estimated as follows. 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 2: The Statistics 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 A.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 1. 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 A.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 2 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 occurring 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 1 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.

Maintaining OLAP Statistics: 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 $PreDim_i$ 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.

B Details of the XML Component Cost Model

This section provides detail on the estimation and maintenance of the OLAP cost parameters.

The choice of which probing 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. Several optimizations are possible, including the retrieval of larger parts of the document in a single query [14]. However, probing may still be a problem for very slow XML components, e.g. on the Web. Hence, the default behavior 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 B.1 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.

C Experiment Queries

This section lists the queries used in the performance experiments.

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
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 3: Queries used in the experiments.