Query Optimization For OLAP-XML Federations

Dennis Pedersen
Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E
9220 Aalborg Ø, Denmark
dennisp@cs.auc.dk

Karsten Riis
Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E
9220 Aalborg Ø, Denmark
riis@cs.auc.dk

Torben Bach Pedersen
Department of Computer Science, Aalborg University
Fr. Bajers Vej 7E
9220 Aalborg Ø, Denmark
tbp@cs.auc.dk

ABSTRACT

The changing data requirements of today's dynamic business environments are not handled well by current OLAP systems. Physically integrating unexpected data into such systems is a long and time-consuming process making logical integration, i.e., federation, the better choice in many situations. The increasing use of Extended Markup Language (XML), e.g. in business-to-business (B2B) applications, suggests that the required data will often be available as XML data. This means that logical federations of OLAP and XML databases will be very attractive in many cases. However, for such OLAP-XML federations to be useful, effective optimization techniques for such systems are needed.

In this paper we present novel techniques for query optimization in federations of OLAP and XML databases. The techniques include so-called *inlining* of XML data in OLAP predicates, caching, and pre-fetching. Experiments show that the proposed optimization techniques improve query execution times significantly. Furthermore, the performance of the optimized federation queries is comparable to the performance achieved with physical integration of the data, showing that our federated OLAP approach is indeed a feasible alternative to physical integration.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems*

General Terms

Performance

Keywords

OLAP, XML, database federations, query optimization

1. INTRODUCTION

OLAP systems [16] 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 is sometimes 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 [16]. Thus, logical, rather than physical, integration is desirable, i.e. a *federated* database system [14] is called for. The increasing use of Extended Markup Language (XML) [18], 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. In order for OLAP-XML federations to perform satisfactorily, effective cost-based optimization techniques are needed. However, existing query optimization techniques do not support the special case of OLAP-XML federations satisfactorily.

This paper presents three effective *cost-based* query optimization techniques for OLAP-XML federations. First, *inlining* of literal XML data values in OLAP queries is suggested to reduce the overhead of evaluating queries that uses external XML data for selection. Second, for simple languages, such as XPath, the special queries needed to retrieve data from XML components can only be expressed using a large number of queries. However, the query interfaces of current systems support only XPath or similar languages. Hence, techniques are presented to combine these queries at the cost of retrieving additional data. Third, techniques are presented to allow *caching* and *pre-fetching* of intermediate query results. Used together, these techniques provide dramatic performance improvements for many queries that would otherwise be unfeasible to evaluate in the federated system. The paper presents experiments that show the effectiveness of the optimization techniques, indicating that our federated approach is a feasible alternative to physical integration.

A great deal of previous work on data integration exist, e.g., on integrating relational data [6], object-oriented data [13], and semistructured data [2]. However, none of these handle the issues related to OLAP systems, e.g., dimensions with hierarchies. One paper [12] has considered federating OLAP and object data, but does not consider cost-based query optimization. The same paper briefly mentions the *inlining* technique, but only for certain simple types of predicates, whereas we consider a cost-based use of the technique. Query processing and optimization has been considered for data warehousing/OLAP systems [16], federated, distributed, and multi-databases [1, 14, 19], heterogeneous databases [5], data sources with limited capabilities [3, 4], and XML and semistruc-

DOLAP'02, November 8, 2002, McLean, Virginia, USA.

tured data [2, 7]. However, none of this work addresses the special case of optimizing OLAP queries in a federated environment.

We believe this paper to be the first to consider query optimization for OLAP-XML federations. Specifically, we believe the cost-based inlining technique and XML retrieval technique, and our particular use of caching and pre-fetching, to be novel. We also believe that optimization issues for federations involving OLAP databases have not been investigated to this extent before.

The rest of the paper is organized as follows. Sections 2 and 3 motivates the use of OLAP-XML federations and introduces the basic concepts of these, respectively. Section 4 presents the architecture of the federation system and gives an overview of query processing and the optimization techniques. Section 5 presents the cost-based technique for inlining XML data in OLAP predicates. Sections 6, 7, and 8 discusses the techniques for limited XML query interfaces, the use of caching and prefecthing, and the combined use of the three techniques, respectively. Section 9 presents the performance experiments conducted to evaluate the effectiveness of the optimization techniques. Section 10 summarizes the paper and points to future work.

2. MOTIVATION

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, enabling queries on this external data. Thus, 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, an OLAP database containing data about products and their production prices can be enriched with information a competing company's products and prices, residing on the competing company's website. 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.

Our approach, a so-called *loosely coupled federation* [14], provides the ability to do *ad hoc integration*, which is 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 the data sources are outside the organisation controlling the federation, e.g., when a component is accessed on the Internet. Also, data is always *up-to-date* when using a federated system as opposed to physically integrating the data, which is crucial for dynamic data such as price lists and stock quotes.

3. OLAP-XML FEDERATION CONCEPTS

This section briefly describes the concepts underlying the OLAP-XML federations that the optimizations are aimed at. The examples are 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 XML document has the following nesting of elements and attributes: Components(Supplier(Class(Component(Manufacturer <@MCode> UnitPrice Description)))). The details of the concepts and the case study are described in another paper [9].

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 [8] where the hierarchies are not balanced trees, e.g., where a lower-level item has several parents. The data model captures this aspect as it can affect the *summarizability* of the data [8], 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 [8].

Extended Markup Language (XML) [18] 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 [18] 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

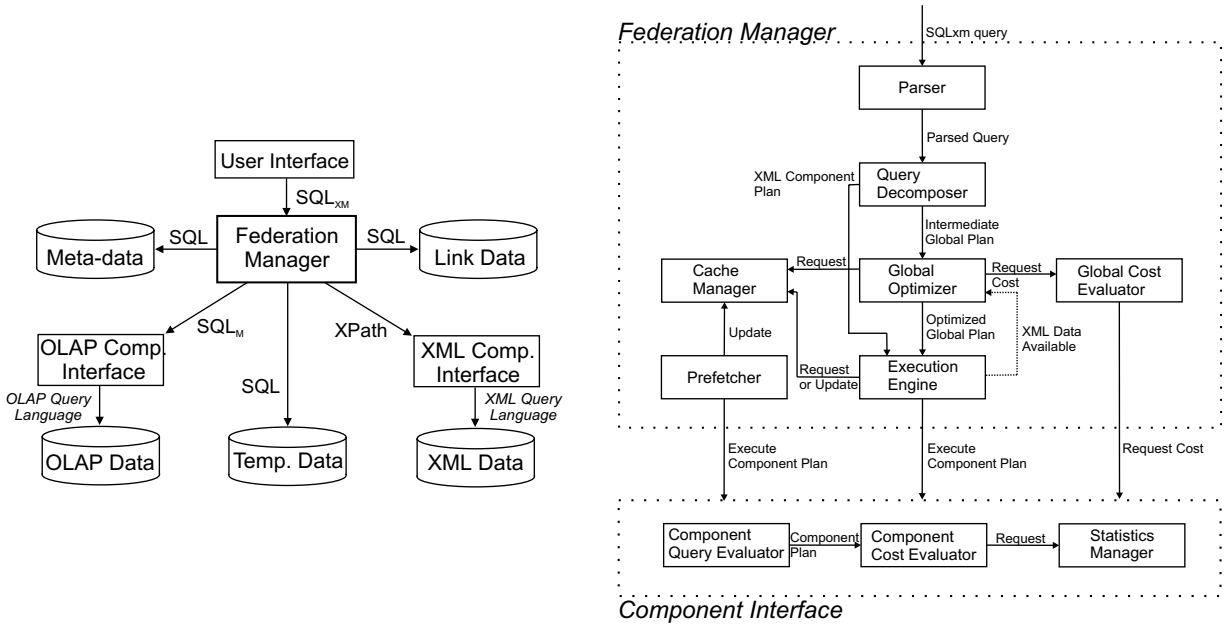

Figure 1: Architecture Of The Federation System and The Federation Manager

in an XML document, e.g., linking electronic components (ECs) in the cube to the relevant "Component" elements in the XML document. The most common type is the *natural link* where the dimension values are also found as element values in the XML document. *Enumerated links* map each dimension value explicitly to a specific XML element 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 so-called *level expressions* of the form *level/link/xpathexpression*, e.g., "*EC/EClink/Manufacturer/@MCode*", in the SELECT, WHERE, and GROUP BY clauses of a query. The link may be omitted in the level expression, in which case the *default link* is used. 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.

4. THE FEDERATION SYSTEM

In this section we give an overview of the OLAP-XML federation system and some design considerations, and introduce the optimization techniques. The overall architectural design of the partially implemented prototype system supporting the SQL_{XM} query language is seen to the left in Figure 1 (the right side of the figure will be explained later). 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 do not support irregular dimension hierarchies and make it too expensive to add new dimensions, 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 [15], which provides an XPath-like interface. For the external temporary component, a leading industrial RDBMS system is used.

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. See [9] for details on the query processing. Since the primary bottleneck in the federation will most often be the moving of data from OLAP and XML components, our optimization efforts have focused on this issue. Three different *cost based* optimization techniques are presented: *inlining XML Data*, *XML data retrieval through limited query interfaces*, and our particular use of *caching and pre-fetching*.

5. INLINING XML DATA

The OLAP and XML components can be accessed in parallel if no XML data is used in the construction of OLAP queries. 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. However, often it is advantageous to make the OLAP query depend on the XML queries, as will be explained next. A level expression such as "EC/Description" can be integrated into a predicate by creating a more complex predicate that contains only references to constants, a process we have termed *inlining*. In the general case the predicate can be very large, but for many simple predicates and if the number of values is small, this is indeed a practical solution. 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 unless inlining is used. Inlining level expressions may also be a good idea if it results in a more selective predicate.

EXAMPLE 5.1. *The level expression predicate EC/Description = '16-bit flip-flop' can be transformed to EC IN (EC1234, EC1235) because EC1234 and EC1235 are the ECs with Description nodes equal to "16-bit flip-flop" (See the full paper [11] for details on the case study.).*

Level expressions can be inlined compactly into some types of predicates. Even though it is always possible to make this inlining, 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. 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. Please refer to the full paper [11] for an in-depth discussion of the above. A 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. 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.

EXAMPLE 5.2. *Consider the predicate: "EC/Manufacturer/@MCode = Manufacturer(EC)". The decoration data for the level expression is retrieved from the XML document as explained in Section 4 resulting in the following relationships between dimension values and decoration values:*

EC	*Manufacturer/@MCode*
EC1234	*M31*
EC1234	*M33*
EC1235	*M32*

Using this table, the predicate can be transformed to: "(Manufacturer(EC) IN *(M31, M33)* AND *EC='EC1234')* OR *(Manufacturer(EC)* IN *(M32)* AND *EC='EC1235')". This predicate is then put into the WHERE clause of the OLAP query. If the predicate is too long, it can be split into: "Manufacturer(EC)* IN *(M31, M33)* AND *EC='EC1234' " and "Manufacturer(EC)* IN *(M32)* AND *EC='EC1235' ".*

Because of the typically high cost of performing extra queries, the cost model is 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. Due to space constraints, we cannot give the details of the cost model and the estimation of cost model parameters here, they are described in detail in another paper [10].

EXAMPLE 5.3. *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.*

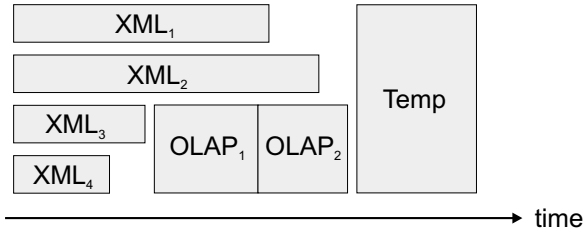

Figure 2: Component Evaluation Times for a SQL_{XM} Query

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. Although predicates in a query will typically contain only a few level expressions, a higher number is, of course, possible. If the number of level expressions n is too high, then the optimal solution cannot be found within reasonable time. In fact, the problem of finding the minimal cost is *NP-hard* as stated in the following theorem.

THEOREM 5.1. *Finding the global minimum of the cost function $Cost$ is NP-hard.*
Proof: *See the full paper [11]*

Even though there is an exponential number of inlining strategies, this will almost never be a problem, as selections typically contain only a few level expressions, say 2–5. Thus, performing an exhaustive search is in most cases an adequate solution to this problem. A few heuristics are used to reduce the search space even more: If a combination of level expressions produces a cost that is much larger than the combinations previously generated, it is not used to generate further combinations. For this to be a valid heuristic, it is important to begin from the full set of level expressions and remove each one iteratively until none are left. Using the opposite approach, i.e. beginning from the empty set and add elements iteratively, the heuristic will sometimes fail. Consider e.g. the two approaches shown in Figures 5 and 5, where each letter represents a level expression, each letter combination represents the choice of inlining these level expressions, and the values shown for some of the combinations represent costs. The optimal results are shown in dashed circles. A top-down approach, beginning from the empty set of expressions, is shown in Figure 5 and a bottom-up approach, beginning from the full set, in Figure 5. A heuristic top-down approach ignores all combinations containing a certain level expression if it is very expensive to inline. For instance, all combinations containing B would be ignored in Figure 5. However, even though an expression is much more expensive to inline than the other expressions, this may be compensated for by a much higher level of aggregation in the OLAP query. This may only be achieved when the expensive level expression is inlined together with other expressions as discussed above. Hence, the optimal cost, represented by the dashed circle, may actually contain expression B, and consequently, the top-down approach will fail to find the optimal value. A bottom-up approach is less likely to fail because it first considers the strategy where all expressions are inlined. Thus, if any combinations give a special reduction in the cost, this is taken into account when selecting which combinations to disregard.

Certain situations may still cause the bottom-up approach to fail, but a refinement to the heuristic can often prevent this. Consider the situation in Figure 5. Here, the optimal strategy may be skipped because all the combinations leading to it are very expensive. This can only occur if all the level expressions not in the optimal strategy

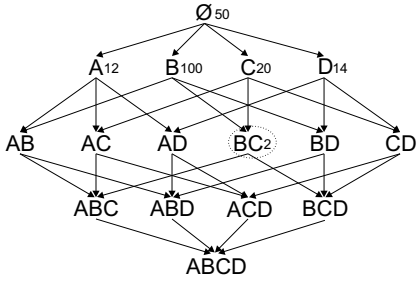

Figure 3: Top-down generation of inlining strategies.

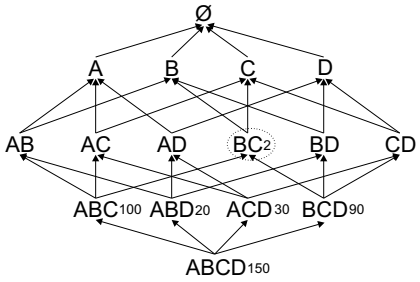

Figure 4: Bottom-up generation of inlining strategies.

take very long time to evaluate or produce a high number of OLAP queries. In Figure 5, both A and D would have to be very expensive. First, this makes it likely that the other combinations, which also involve A or D, are expensive too. Second, the heuristic can be refined to handle this problem by identifying when a single level expression causes a long waiting time or many OLAP queries.

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

6. OPTIMIZING XML DATA RETRIEVAL

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. Having only an XPath interface, decoration data can be fetched by posing a query for each dimension value. However, in general the overhead of transferring, parsing and evaluating a large number of queries will be too expensive. Combining groups of queries into a single query using OR/IN predicates (like Example 5.2) does not work because the resulting nodes cannot always be distinguished in

the result. For this to be possible, both the locator and the user defined XPath expression must be present. This is necessary because according to the XPath Recommendation the result of an XPath expression is an *unordered* collection of nodes, and hence, there is no way to identify each decoration node, except by using the locator. Furthermore, an XPath expression cannot change the structure of a document, but only return a set of existing nodes from it. Consequently, the result of an expression can only contain *entire* nodes and not partial nodes. Thus, to maintain the relationship between the locator and the user defined XPath expression, their common parent node must be retrieved in its entirety. 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. If the level expression is based on a natural link, it is always possible to combine all the expressions which typically produces a low overhead expression because of the way these links are defined. If it is based on an enumerated link, combining all expressions to a single expression may retrieve all or a large part of the document. Hence, we also consider the situation where only expressions having predicates at the same location step are combined. For each of these three strategies, the total evaluation cost is estimated and the cheapest one is used. The details of the technique, including an algorithm for combining XPath expressions is described in the full paper [11].

7. CACHING AND PRE-FETCHING

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

Determining whether a cached result can be used to produce a new result must be done efficiently since a large cache may hold many different results. This is done by performing a few simple tests on the query tree corresponding to each cached result. Consider two query trees Q_C and Q_U, representing the *cached* and the *user* query, respectively. If the cached result can be used to produce the needed result then Q_U must be expressible in terms of Q_C. The tests discussed next will determine whether this is possible, but let us first see how the final query is constructed in the non-trivial case where Q_U and Q_C are not identical. (By identical we mean that they contain the same operations, disregarding the order of selections.) Q_C must form the bottom part of the *new* query Q_N that is constructed from Q_U, and the part of Q_U that restricts the result further must be applied to this bottom part (see the full paper [11] for details).

In summary, we perform caching and pre-fetching for component queries only. Intermediate OLAP results stored in temporary tables as well as raw XML data are kept for a certain amount of time, which can specified as a tuning parameter. If adequate storage is available, temporary XML tables are also stored to avoid constructing the same tables again. Currently, we do not cache or pre-fetch entire federation queries as the cache space they take up is in most cases too high in comparison with the probability that they can be re-used. A *least recently used* replacement strategy is used. As will be described in Section 9, experiments indicate that large cost reductions can be achieved using these techniques. This technique is described in detail in the full paper [11].

8. COMBINING THE TECHNIQUES

We now outline how the techniques discussed above are used in combination, we will refer to the software component architecture seen to 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. Here, the specific actions depend on which query languages are supported. If e.g. only an XPath interface is available, it is determined which is cheaper: To pose many queries or to combine some or all of them into a single query. This decision is based on costs estimated by the Component Cost Evaluator. For the OLAP part of the query, it is also determined whether the result can be obtained from any of the cached results. If so, the cost of evaluating the SQL_{XM} query using the cached results is compared to the cost of evaluating the SQL_{XM} query without the use of cached results. This cost is determined by the Global Cost Evaluator. The cost estimates are also used by the Global Optimizer to pick a good inlining strategy. When the results of the component queries are available in the temporary component, the relational part of the SQL_{XM} query is evaluated.

9. EXPERIMENTAL RESULTS

This section describes the experiments performed to evaluate the effectiveness of the optimization techniques. The test cube is based on about 50 MB of data generated using the TPC-H benchmark [17] 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. The details of the experiments, including the actual queries, can be found in the full paper [11].

The results of executing the queries are shown in Figure 5. The results only show 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) may be performed several times for one query) : *Task 1(a)*: fetch XML data and store it in the temporary component (denoted by a prefix "X", e.g., "X1"); *Task 1(b)*: fetch OLAP data and store it in the temporary component (denoted by an "O"); and *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. As the prototype is not yet fully implemented, the query optimization process has been carried out by hand. Thus, the execution costs does not contain the cost of optimizing the query. However, this cost will be neglible compared to the high cost of executing the OLAP and XML queries.

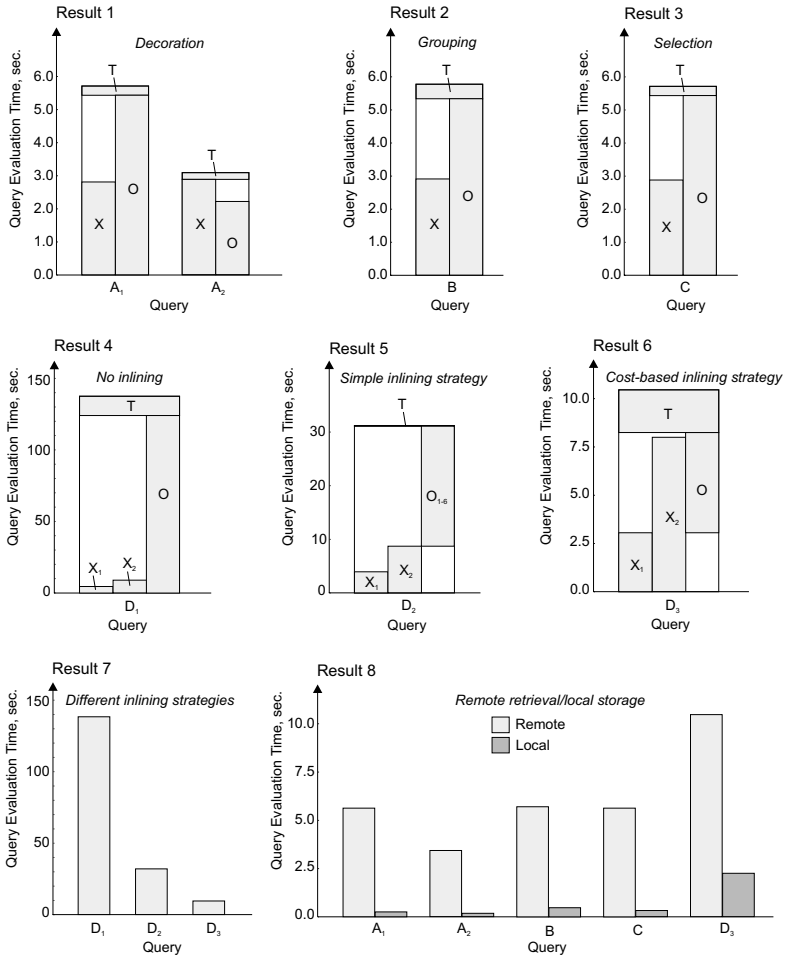

Figure 5: Experimental results. Notice that different time intervals are used in the graphs.

Query A_1 and A_2 both illustrate the use of decoration. The results of these queries, seen in Result 1, shows a low overhead of performing decoration compared to the time it takes to retrieve the OLAP and XML data. This low overhead is representative for decoration queries since they do not require additional data to be retrieved from the OLAP component. Hence, the overhead of combining the intermediate results will be low, since typically only small amounts of OLAP data (at most a few thousand facts) will be requested for presentation to a user. Query B illustrates the use of XML data for grouping, while Query C uses XML data for selection. Since both queries use the same decoration as in Query A_1, the same XML and OLAP queries are performed to evaluate these queries. As can be seen from Result 2 and 3, the overhead endured by the temporary component query is also low for these queries. This is typical for both grouping and certain types of selection because the size of the intermediate OLAP result will often be comparable to the size of the final result.

Results 4, 5, and 6 demonstrate how effective the inlining technique can be. Here, two XML queries are used: X_1 is the time it takes to retrieve the Population data, while X_2 is the time it takes to retrieve the RetailPrice data. The results show the processing times for three different strategies for evaluating Query D, in which the selection on Population refers to a low level that is not needed in the result. Thus, in Result 4, where no inlining is used, this produces a very large overhead as the OLAP query can only aggregate to a much lower level than needed for the final result. Result 5 illustrates the use of the simple inlining strategy, "Always use inlining if the predicate is simple", which causes both XML predicates to be inlined. This is significantly faster than before because the OLAP query can now aggregate further. 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 and the OLAP query has to wait for the XML queries to finish. Consequently, in this example, it is faster to inline only the Population data. This is shown in Result 6 where the cost-based inlining strategy is used. Because the Nation level only contains few dimension values, this requires only one OLAP query and the Population data is also faster to retrieve. The three results are also shown in Result 7 for easy comparison. We see that cost-based inlining improves the performance by more than an order of magnitude.

To evaluate the effectiveness of caching and prefetching, the queries have also been evaluated using a fully pre-aggregated cube where both the OLAP data and the XML data were stored in tables. These *local* evaluation times are shown in Result 8 together with the non-local, or *remote*, evaluation times described above. We see that the use of cached/prefetched data improves performance from 4 to 25 times in this case. However, more experiments are needed to fully understand the effect of caching and prefetching.

To evaluate the combined effect of the query processing and op-

timization techniques for our federation, 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 6. The "O" bars show the time spent in the OLAP component, while the "T" bars show the time spent in the temporary component. The queries A_1, B, and C from above were used in the comparison. The figure shows that for typical queries the overhead of our federated 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.

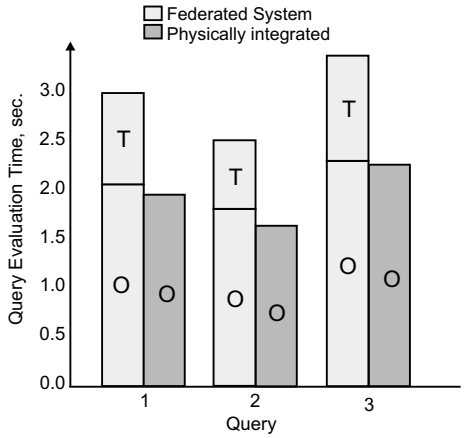

Figure 6: Federation Versus Physical Integration

10. CONCLUSION AND FUTURE WORK

Motivated by the need for federating external XML data with OLAP data, and the subsequent need for efficient query processing in such OLAP-XML federations, we have presented three effective *cost-based* query optimization techniques for such systems. First, *inlining* of literal XML data values in OLAP queries was suggested to reduce the overhead of evaluating queries that uses external XML data for selection. Second, for XML components offering only a simple query interface, e.g., XPath, techniques to combine these queries at the cost of retrieving additional data were presented. Third, techniques were presented to allow *caching* and *pre-fetching* of intermediate query results. These techniques provide dramatic performance improvements for many queries that would otherwise be unfeasible to evaluate in the federated system. The paper also presented experiments that show the effectiveness of the optimization techniques. Additionally, the experiments indicate that our federated approach is indeed a practical alternative to physical integration.

We believe this paper to be the first to consider query optimization for OLAP-XML federations. Specifically, we believe the the cost-based use of the inlining technique and the XML retrieval technique, and our particular use of caching and pre-fetching, to be novel. We also believe that optimization issues for federations involving OLAP databases have not been investigated to this extent before.

Future work will focus on implementation aspects, e.g., exploring how this work could be applied in a commercial software product such as an existing OLAP querying tool. Here, interesting research issues include how to capture the document order of an XML document in the result of an OLAP query, how to extract new measures from XML data and incorporate these into a cube, data cleansing for this particular setting, and finally optimization techniques for all these issues.

11. REFERENCES

[1] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *Proc. of SIGMOD*, pp. 137–048, 1996.

[2] S. Chawathe et al. The TSIMMIS project: Integration of Heterogeneous Information Sources. In *Proc. of the 16th Meeting of the Information Processing Society of Japan*, pp. 7–08, 1994.

[3] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query Optimization in the Presence of Limited Access Patterns. In *Proc. of SIGMOD*, pp. 311–022, 1999.

[4] H. Garcia-Molina and R. Yerneni. Coping With Limited Capabilities Of Sources. In *Proc. of BTW*, pp. 1–09, 1999.

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

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

[7] J. McHugh and J. Widom. Query Optimization For XML. In *Proc. of VLDB*, pp. 315–026, 1999.

[8] 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. 121–030, 2002.

[9] D. Pedersen, K. Riis, and T. B. Pedersen. XML-Extended OLAP Querying. In *Proc. of SSDBM*, 2002.

[10] D. Pedersen, K. Riis, and T. B. Pedersen. Cost Modeling and Estimation for OLAP-XML Federations. In *Proc. of DaWaK*, pp. 245–254, 2002.

[11] D. Pedersen, K. Riis, and T. B. Pedersen. Query Optimization For OLAP-XML Federations. TR R02-5004, Department of Computer Science, Aalborg University, 2002, `www.cs.auc.dk/~tbp/publications`

[12] T. B. Pedersen, A. Shoshani, J. Gu, and C. S. Jensen. Extending OLAP Querying To External Object Databases. In *Proc. of CIKM*, pp. 405–413, 2000.

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

[14] 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.

[15] Software AG. Tamino XML Database. `www.softwareag.com/taminoplatform`, 2001. Current as of July 5, 2002.

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

[17] Transaction Processing Performance Council. TPC-H. `www.tpc.org/tpch`, 2001. Current as of July 5, 2002.

[18] W3C. Extensible Markup Language (XML) 1.0 (Second Edition). `www.w3.org/TR/REC-xml`, October 2000. Current as of July 5, 2002.

[19] Q. Zhu and P.-Å. Larson. Global Query Processing And Optimization In The CORDS Multidatabase System. In *Proc. of PDCS*, pp. 640–646, 1996.