AALBORG UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
FREDRIK BAJERSVEJ 7 - DK 9220 AALBORG - DENMARK

Title:

Building an OLAP-XML Query Engine

Project period:

2003.01.02 – 2003.12.06

Project group:

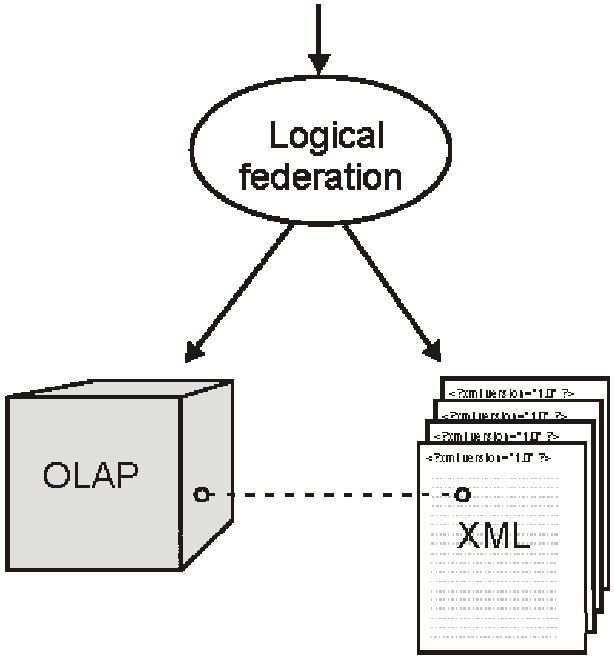
E3-217

Members:

Xuepeng Yin

Advisor:

Torben Bach Pedersen



Report 1

Pages: 71

Copies: 5

Preface

1106, 2003, Aalborg University

This project is written at the KDE4 semester 2003,

The main purpose of the project is to extend the logical OLAP-XML federation with practical optimization techniques, implement an optimized OLAP-XML query engine, and carry out performance experiments.

I would like to acknowledge the generousity of my supervisor, Torben Bach Pedersen, for his great help and comments in preparing this paper.

Xuepeng Yin

Contents

Abstract

In today's OLAP system, integrating fast changing data (e.g. stock data) physically into a cube could be complex and time-consuming, The XML technology today makes it very possible that this data is available in XML databases. Thus, making XML data logically federated into OLAP systems is greatly needed. In previous work, an approach to the logical OLAP-XML federation has been developed, which includes the data models, a SQL_{XM} query language, querying techniques, physical algebra and a prototype of OLAP-XML query engine. In this paper, the OLAP-XML Query Engine now has been integrated with practical query optimization techniques. A rule-based and cost-based query optimizer generates a plan space given the initial plan, and selects the least cost plan in the plan space. New operators and transformation rules are introduced to generate plans. Pruning techniques such as Branch and Bound are used to reduce the plan space. An optimization technique, *inlining*, is integrated in physical algebra and supported by the new OLAP-XML query engine. Experiments about effectiveness of query optimizations techniques are performed on the new engine, indicating that the optimizations have effectively boosted up the querying performance and greatly improved the logical federation of OLAP and XML systems.

1 Introduction

More and more mature systems for On-line Analytical Processing (OLAP) or Extensible Markup Language (XML) emerge these days. OLAP technology enables data warehouses to be used effectively for online analysis, providing rapid responses to iterative complex analytical queries. Usually an OLAP system contains a large amount of data, but some of the *dynamic data* today, e.g. stock prices, is not handled well in current OLAP systems. To an OLAP system, a well designed dimensional hierarchy and a large quantity of preaggregated data are the keys. However, trying to maintain these two factors when integrating the fast changing data physically into a cube could be complex and time-consuming, or even impossible. Nevertheless, the XML technology today makes it very possible that this data is available in XML databases. Thus, making XML data accessible to OLAP systems is greatly needed. However, the kind of system that supports the real querying over the integration of the data from these two database technologies has not shown up yet.

Our solution is to logically federate the OLAP and XML data sources. This approach allows external XML data to be used as "virtual" dimensions. An OLAP database can be *decorated* with external dimensions, which means OLAP data can be presented along with XML data as a result of a query. *Selection* can be performed over the federation, which means qualifications of XML data can be also put into a query that queries OLAP data. OLAP data may be *grouped* based on external XML data, numeric values can then be *aggregated* over these groups.

In this paper, extensions have been made to the logical OLAP-XML federation systems developed in [21, 20, 19] and the OLAP-XML query engine in [26]. Optimization techniques are

defined as operators and other new operators are integrated into the physical algebra. The query engine is extended with a cost-based and rule-based optimizer. Practical transformation rules and cost formulas for each operator are developed. A prototype of a query engine with optimizations is implemented. The paper presents experiments that show the effectiveness of the optimization techniques, indicating that the optimized OLAP-XML query engine now has made a large step forward towards a mature application and become a practical alternative to physical integration.

There has been a great deal of previous work on data integration, for instance, on relational data [14, 5, 7], semi-structured data [3], and a combination of relational and semi-structured data [13, 16]. However, none of these handle the advanced issues related to OLAP systems, e.g., automatic aggregations and correct aggregation. There is some work on integrating OLAP systems with other systems, such as [22, 12] on integrating OLAP, and object database which demands rigid schemas of data, e.g., data is represented by classes and connected by complex associations. In comparison, Using XML as one component enables the federation to be applied on any data as long as the data allows XML wrapping. The papers related to our topic are [21, 20, 19, 26]. [20, 19] present a logical federation of OLAP and XML systems, but none of them really build up a robust query engine or the physical algebra that supports a running federation system. However, the manager of the federation is not clearly described, and the processing phase of a query is only limited to high-level logical algebra. [21] presents a cost model and query optimization techniques of the federations based on a logical design of the federation system, and does not integrate the strategies on a running query engine. [26] presents a physical algebra of the query language and the implementation of a query engine with a query parser and evaluator, which can execute queries over the federation but does not have a good performance. Our work integrates the query engine [26] with efficient optimizations and significantly reduces the evaluation time, also the experiments in this paper were carried on the running query engine to show the optimization efficiency.

This paper contributes several novelties to the application areas for federations of OLAP and XML systems. New operators are developed, physical algebra of SQL_{XM} queries are revised. Practical transformation rules are developed. Cost estimates are achieved by practical cost formulas and statistics. Moreover, these new features are all implemented in the new optimized query engine.

The rest of the paper is organized as follows. Section 2 presents the system architecture and the case study. Section 3 gives a brief review of the data models used in the federation. Section 4 presents the operators including the new operators and those with revised semantics. Section 5 presents the query semantics with new operators and an example of a query plan. Section 7 presents the transformation rules. Section 9 presents the required statistics for cost estimation and the cost formulas for all the operator. Section 6 describes the structures of the query plan rewriter and the planner. Section 10 presents the implementation of several functions, statistics retrieval and gives out the schematic pseudo-code. Section 11 presents the performance experiments conducted to evaluate the effectiveness of the optimization techniques. Section 12 summarizes the paper and

points to future work.

2 Motivation

In this section, we discuss the federation system with integrated optimization techniques and present a real-world case study that is used for illustration throughout the paper

2.1 Case Study

The example database used in the experiments and illustrations is shown in Figure 1. The database is based on the data generated using the TPC-H benchmark [24].

An OLAP database is a multidimensional database. It contains a set of dimensions and measures. A dimension is an organized hierarchy of levels that describe data in the *fact table*. These levels typically describe a similar set of members upon which the user wants to base an analysis. Measures are the central values that are aggregated and analyzed. A fact table is a central table that contains numerical measures and keys relating *facts* to dimension tables.

The OLAP database, called TData, is characterized by a `Supplier` dimension, a `Parts` dimension, an `Order` dimension, a `LineItems` dimension and a `Time` dimension. `Quantity`, `ExtendedPrice` (`ExtPrice`), `Tax`, `Discount` are measured and stored in the fact table, `linefacts`. Schema of the TData is shown in Figure 2. Example data that are used for illustrations throughout this paper are shown in Figure 3 .

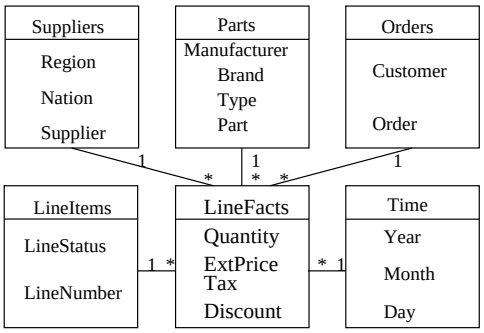


Figure 1: UML schema for the TData OLAP database

A lot of information today are available in XML. The fundamental part of an XML document is the element. Elements are identified by a *start tag* and *end tag*, and can contain other elements, text data, and attributes. For a more comprehensive explanation of XML see [25].

The XML documents we use are shown in Figure 4. One of the document, `Nations`, is about the population of nations. The other, `Types`, is generated from the TPC-H data about the retail-prices of types. Example data are also shown in Figure 4, which will be used in the following

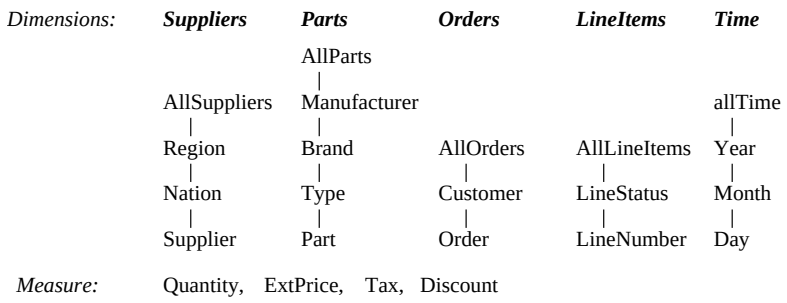


Figure 2: Schema of example cube

illustrations.

The XPath language is chosen to address parts of an XML document. The basic syntax of an XPath expression resembles a Unix file path. An XPath expression selects a set of nodes relative to the *context node* [25]. e.g. "/Nations/Nation[NationName='DK']/Population", selects the population node in the context of the node whose child node has the content "DK".

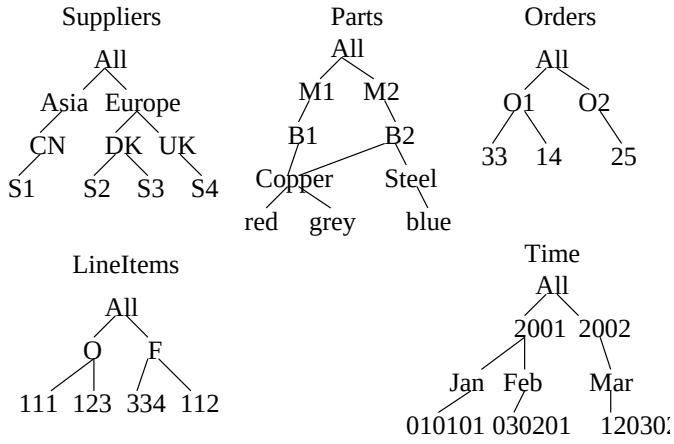


Figure 3: Example data of TData

2.2 Optimized OLAP-XML Query Engine

Optimizations are needed to speed up the query engine. In previous work [26], the OLAP-XML query engine only contains a query analyzer and evaluator. An user query is turned into a plan by the query analyzer and passed directly to the evaluator. The plan is straightforward and experiments have shown that the query engine does not give a good performance. The most significant way to improve the total evaluation time will generally be to reduce temporary data, which reduces data transfer costs, the time it takes to store temporary data, and the time required to produce the final result. One of the *cost-based* query optimization techniques proposed in [21] is *inlining* literal XML

```
<Nations>
    <Nation><NationName>DK</NationName><Population>5.3</Population></Nation>
    <Nation><NationName>CN</NationName><Population>1264.5</Population></Nation>
    <Nation><NationName>UK</NationName><Population>19.1</Population></Nation>
     . . .
</Nations>
 <Types>
     <Type><TypeName>Brass</TypeName><RetailPrice>1890</RetailPrice></Type>
     <Type><TypeName>Copper</TypeName><RetailPrice>1240</RetailPrice></Type>
     <Type><TypeName>Steel</TypeName><RetailPrice>1410</RetailPrice></Type>
     . . .
 </Types>
```

Figure 4: Part of the XML document

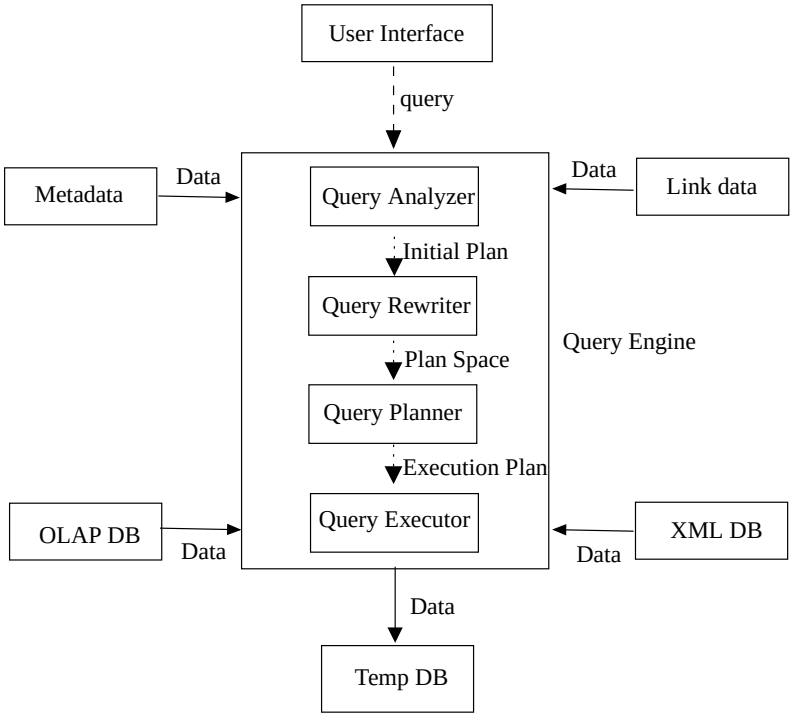


Figure 5: The Architecture of the Optimized Query engine

data values in OLAP queries to reduce the overhead of evaluating queries that uses external XML data for selection. All of these optimizations are taken into concern in the second version and it is believed that it will give a promising performance.

As shown in Figure 5, the query engine has 4 components, query analyzer, query rewriter, query planner and query evaluator. Given an user query, the query engine parses, analyzes the query, and generates the initial plan. The query language is XML extended Multidimensional SQL (SQL_{XM}), which has the basic sublauses from SQL, i.e., SELECT, FROM, WHERE, GROUP BY, HAVING, and the extended expressions for component data references (Section 5). Below is an example SQL_{XM} query, where "Nation(Supplier)" refers to non-bottom dimension level *Nation*, "Nation/NLink/Population" refers to decoration values from XML documents.

```
SELECT      SUM(Quantity),SUM(ExtPrice),Nation(Supplier),
            Supplier
FROM        TData
WHERE       Nation/NLink/Population<50
GROUP BY    Nation(Supplier),Supplier
```

The plan consists of all the steps of the query processing, e.g., the retrieval of data, the operations over the data in temporary database. One new component, the query *rewriter*, is *rule-based* and *cost-based* which means *transformation rules* are used to generate equivalent plan expressions, and *cost-based pruning* is used to prune the plan space. It accepts the initial query plan and then matching plans are found and applied to generate a *plan space* where the candidate execution plans reside. The *plan space* is then passed to the other new component, the query *planner*. The *planner* is *cost-based* which selects the cheapest plan in cost. The last component is the query *evaluator* which follows the operators in the given plan and gives the final result. Generally, component data is first retrieved from OLAP and XML data sources. If the plan is optimized with the *inlining* technique, the to-be-inlined XML data will be retrieved first of all and then inlined. Other OLAP and XML data then can be retrieved in parallel to the temporary component. For an optimized plan, the underlying OLAP cube may be sliced and aggregated which leads to less OLAP data to be transfered. Then, temporary operations, i.e. SQL operations, are performed on the gathered data from different data sources. OLAP data are decorated when external dimension containing both OLAP dimension values and decoration values are built. OLAP data are sliced when SQL selections with conditions on external dimensions are evaluated. Numeric data is aggregated over the grouped decoration data and temporary dimension tables. After a series of temporary operations, the final result is given in the temporary database.

3 Data Models

we give a brief review of the data models that are originally written in [26].

A *cube*, C, is the underlying OLAP database. $C = (N, D, E, F)$, where N is the cube's name,

D is a set of dimensions, E is a set of dimension values, F is the fact table. Figure 6 shows schematically a fact table. A measure is denoted using M_i. A dimension D_i has a hierarchy of the

Quantity	ExtPrice	Tax	Discount	Supplier	Part	Order	LineNumber	Day
17	17954	0	0	S1	red	14	334	010101
36	73638	0	0	S2	grey	33	112	030201
28	29983	0	0	S2	blue	25	123	120302
26	26374	0	0	S4	blue	14	111	030201
2	2388	0	0	S1	red	25	111	030201

Figure 6: A schematic fact table

levels, LS_i. A level l_{ij} is the j-*th* level in LS_i. $\perp_i$ is the unique bottom level, while $\top_i$ is the unique "ALL" level. Each pair of levels has a containment relationship, that is, the *partial order* of two levels, $l_{i1} \sqsubseteq_i l_{i2}$, which holds if elements in $L_{l_{i2}}$ can be said to contain the elements in $L_{l_{i1}}$. $L_{l_{ik}}$ is the dimension values of level l_{ik}. Similarly, we say that $e_1 \sqsubseteq e_2$ if e_1 is logically contained in e_2 and $l_{ij} \sqsubseteq_i l_{ik}$ for $e_1 \in L_{l_{ij}}$ and $e_2 \in L_{l_{ik}}$ and $e_1 \neq e_2$. A *temporary cube*, $\mathcal{C}$, is a partial copy of cube C with external decoration dimensions. A temporary cube resides in a relational database. It is a four-tuple (N, D, E, F, T), where the cube name N, a non-empty set of dimensions D, a set of dimension values E, the fact table F which is used to hold the facts data transfered from the OLAP cube, and a set of temporary dimension tables $T = \{R_1, \ldots, R_t\}$. *Federation* is the data structure on which we perform the data manipulation operations, e.g. selections, aggregations and decorations. It includes the OLAP cube, XML documents, and a temporary cube. A federation $\mathcal{F}$ of a cube C and a set of XML document X is a four-tuple: $\mathcal{F} = (C, Links, X, \mathcal{C})$ where $Links$ is a set of links between levels in C and documents in X, $\mathcal{C} = \{N, D, E, F, T\}$ is a temporary cube of C.

A roll-up expression is $l_{ij}(l_{ik})$, where $l_{ij} \sqsubseteq_i l_{ik}$. An XPath expression xp is a path that selects a set of nodes relatively in the context of s. $AbsXP_x$ is the absolute XPath expression that starts with the root node of the document, while relative XPath expressions ($RelXP_x$) starts with a given context of some node. A link is a relation $Link$ that connects dimension values with nodes in XML documents. There are two kinds of link. An *enumerated link* $EnLink : \mathcal{P}(L_1 \times X \times AbsXP_x)$ connects dimension values in L_1 with the nodes in the set of XML documents X by absolute paths. An *natural link* $NatLink : \mathcal{P}(L \times X \times AbsXP_x \times RelXP_x \times Alias)$ maps dimension values in L to the $Alias$ of the string values of the nodes selected by $AbsXP_x$ and $RelXP_x$. For example, a natural link, NLink = ("Nation", "nations.xml", "/Nations/Nation/", "NationName") connects the nodes pointed at by "Nations/Nation" with the dimension values identical to the value of "NationName" nodes. A level expression $L/link/xp$ defines a link between L and the nodes selected by xp in the context of the nodes in a link $link$. To ensure correct aggregation, an *aggregation type* is associated to each combination of a measure and a dimension. Three types of data are distinguished: c, data that may not be aggregated because summarizability is not preserved,

ϕ, data that may be averaged but not added, and Σ, data that may also be added. A function AggType:$\{M_1, \ldots, M_m\} \times D \mapsto \{\Sigma, \phi, c\}$ returns the aggregation type of a measure M_j when aggregated in a dimension $D_i \in D$.

4 Operators

In this section, we give the definitions of operators.

4.1 Fact-Transfer

The fact-transfer operator, ϕ, builds a temporary cube and loads the facts selected by cube operators into the temporary cube.

Definition 4.1 (Fact Transfer) Let $\mathcal{F}$ be a federation that is $\{C, Links, X, \mathcal{C}\}$, where $\mathcal{C} = (N, D, E, F, T)$, $F = \emptyset$, $T = \emptyset$. Facts-transfer operator is defined as:

$$\phi(\mathcal{F}) = \mathcal{F}'$$

where $\mathcal{F}' = (C, Links, X, \mathcal{C}')$, $\mathcal{C}' = (N, D, E, F', T)$, F' is the temporary table containing facts.

Example 4.1 For $\phi(\mathcal{F})$, the fact-transfer leads to the fact table for the example TData database shown in Figure 6 being loaded into the temporary cube.

4.2 Dimension Transfer

Dimension-transfer operator, ω, copys the dimension values for the given dimension levels into a temporary table.

Definition 4.2 (Dimension Transfer) Let $\mathcal{F}$ be a federation that is $\{C, Links, X, \mathcal{C}\}$, where $\mathcal{C} = (N, D, E, F, T)$, Dimension-transfer operator is defined as:

$$\omega_{(l_{ix}, l_{iy})}(\mathcal{F}) = \mathcal{F}'$$

where $\mathcal{F}' = (C, Links, X, \mathcal{C}')$, $\mathcal{C}' = (N, D, E, F, T')$, $T' = T \cup \{R\}$, $R = \big\{(e_{ix}, e_{iy}) | e_{ix} \in L_{ix} \wedge e_{iy} \in L_{iy} \wedge \forall \{e_1, e_2\} \subseteq \{e_{ix}, e_{iy}\} (e_1 \sqsubseteq_{D_i} e_2)\big\}$, $\{e_{ix}, e_{iy}\}$ is a multiset, $l_{ix} \in LS_i$, ..., $l_{iy} \in LS_i$. $L_{ix}, \ldots, L_{iy}$ represent the dimension values of levels of D_i, LS_i is a set of levels of D_i, $D_i \in D$,$D \in C$.

Since a dimension in a federation needs dimension values to roll up to a specified level, a roll-up expression, $l_i(\perp_i)$, in SQL_{XM} query always yields a dimension transfer operator, $\omega_{[\perp_i, l_i]}(\mathcal{F})$, in a query plan. We denote this behavior with: $l_i(\perp_i) \rightsquigarrow \omega_{[\perp_i, l_i]}(\mathcal{F})$.

Nation	Supplier
CN	S1
DK	S2
DK	S3
UK	S4

Figure 7: The Table-T1

Example 4.2 A roll-up expression, "Nation(Supplier)", yields a dimension transfer, that is $Nation(Supplier) \rightsquigarrow \omega_{[Nation,Supplier]}(\mathcal{F})$, the dimension values for `Supplier` and `Nation` are loaded into a temporary table, T1, shown in Figure 7. $T1$ is put into T as described in the definition.

4.3 XML Transfer

XML-transfer operator, τ, copys the decoration values in the XML documents to the temporary database. The decoration values are the string values of the nodes pointed at by the XPath expressions in level expressions.

Definition 4.3 (XML Transfer) Let $\mathcal{F}$ be a federation that is $\{C, Links, X, \mathcal{C}\}$, where $\mathcal{C} = (N, D, E, F, T)$, XML-transfer operator, τ, is defined as:

$$\tau_{(l_z[SEM]/link/xp)}(\mathcal{F}) = \mathcal{F}'$$

, where $\mathcal{F}' = (C, Links, X, \mathcal{C}')$, $\mathcal{C}' = (N, D, E, F, T')$, $T' = T \cup \{R\}$.

$$R = \begin{cases} \{(e_z, e_{xp}) | (e_z, s) \in link \wedge e_{xp} = strVal(xp(s))\} & if\ SEM = ALL \\ \{(e_z, e_{xp}) | (e_z, s) \in\ some\ link \wedge e_{xp} = strVal(xp(s))\} & if\ SEM = ANY \\ \{(e_z, e_{xp}) | (e_z, s) \in link \wedge e_{xp} = strValueForAll(s))\} & if\ SEM = CONCAT \end{cases}$$

where $strValueForAll(s) = concat(strVal(xp(s'_1)), \ldots, strVal(xp(s'_{|s|})))$, $s'_k \in s$, $k \in \{1, \ldots, |s|\}$, where $|s|$ denotes the number of elements s has.

A level expression always yields an XML-transfer, we denote this with:
$l_z/Link/xp \rightsquigarrow \tau_{[l_z/Link/xp]}(\mathcal{F})$.

Example 4.3 For $\tau_{[Nation/NLink/Population]}(\mathcal{F})$, a temporary table is yielded into the temporary cube containing the decoration values and the dimension values of the starting level, `Nation`. The level expression does not specify the decoration semantics, so the default ALL is used. All the values for the starting levels are covered. The table, T2, is shown in Figure 8. For simplicity, we say the level expression, "Nation/NLink/Population", yields this XML transfer, that is, $Nation/NLink/Population \rightsquigarrow \tau_{[Nation/NLink/Population]}(\mathcal{F})$ R is put into T as described in the definition.

Nation	Population
DK	5.3
CN	1264.5
UK	19.1

Figure 8: The Table-T2

4.4 Decoration

The decoration operator builds the decoration dimension.

Definition 4.4 (Decoration operator) Let $\mathcal{F}$ be a federation that is $\{C, Links, X, \mathcal{C}\}$, where $\mathcal{C} = (N, D, E, F, T)$. The decoration operator of a federation $\mathcal{F}$ is defined as:

$$\delta_{(l_z[SEM]/link/xp)}(\mathcal{F}) = \mathcal{F}'$$

where $l_z \in LS_z$, $link$ is a link from l_z to X, xp is an XPath expression over X.

$\mathcal{F}' = (C, Links, X, \mathcal{C}')$, $\mathcal{C}' = \{N, D', E', F, T'\}$, where $D' = D \cup \{D_{n+1}\}$, $E' = E \cup \{E_{D_{n+1}}\}$, $T' = T \cup \{R_{D_{n+1}}\}$ Suppose R_τ is a relation returned by $\tau_{[l_z[SEM]/link/xp]}(\mathcal{F})$, $R_{D_{n+1}}$ is a relation returned by $\omega_{[\perp_z, l_z]}(\mathcal{F})$, then, since both tables have a common attribute, l_z, the temporary dimension table is a natural join between R_τ and R_ω with attributes $l_\perp$ and l_{xp}, that is, $R_{D_{n+1}} = \pi_{R_{tau}.l_{xp}, R_\omega.l_\perp}(R_\tau \bowtie R_\omega)$. where $D_{n+1} = (LS_{n+1}, \sqsubseteq_{n+1}, \top_{n+1}, \perp_{n+1})$ where $LS_{n+1} = \{\top_{n+1}, l_{xp}, \perp_{n+1}\}$ and $\sqsubseteq_{n+1}$= $\{(\perp_{n+1}, l_{xp}), (l_{xp}, \top_{n+1}), (\perp_{n+1}, \top_{n+1})\}$, $\perp_{n+1} = \perp_z$.

$E_{D_{n+1}}$ is derived from $R_{D_{n+1}}$ and D_{n+1} according to Definition (3.4) in [26]. For each measure M_h in M the aggregation type of D_{n+1} is: AggType(M_h,D_z).

Since a level expression is a SQL_{XM} query always means the cube should be decorated by referenced XML data with specified decoration semantics, that is, a level expression, $l_z[SEM]/Link/xp$, always yields a decoration operator, $\delta_{[l_z[SEM]/Link/xp]}(\mathcal{F})$, we denote this behavior with: $l_z[SEM]/Link/xp \rightsquigarrow \delta_{[l_z[SEM]/Link/xp]}(\mathcal{F})$, where $\mathcal{F}$ is the federation with temporary tables built by $\omega_{[\perp_z, l_z]}(\mathcal{F})$ (in case $l_z \neq \perp_z$) and $\tau_{[l_z[SEM]/Link/xp]}(\mathcal{F})$.

Example 4.4 For $\delta_{[Nation/NLink/Population]}(\mathcal{F})$, the temporary dimension table, new dimension and dimension values are built. The temporary dimension table is shown below.

Supplier	Population
S1	1264.5
S2	5.3
S3	5.3
S4	19.1

Figure 9: The Temporary Decoration Dimension Table

The decoration dimension, D_{n+1}, is ($LS_{n+1}, \sqsubseteq_{n+1}, \top_{n+1}$, Supplier), where LS_{n+1} = {(Supplier,Population), (Supplier, $\top_{D_{n+1}}$), (Population, $\top_{D_{n+1}}$)}. The dimension values, $E_{D_{n+1}}$,

can be generated. $E_{D_{n+1}} = (L_{D_{n+1}}, \sqsubseteq_{D_{n+1}})$, where $L_{D_{n+1}} = \{\top_{D_{n+1}}, S1, S2, S3, S4, 1264.5, 5.3, 19.1\}$. The other component, $\sqsubseteq_{D_{n+1}}$ can be derived from the Cartesian product of $\{(\top_{D_{n+1}})\}$ with the temporary table, which is shown below.

Supplier	Population	ALL
S1	1264.5	$\top_{D_{n+1}}$
S2	5.3	$\top_{D_{n+1}}$
S3	5.3	$\top_{D_{n+1}}$
S4	19.1	$\top_{D_{n+1}}$

then, $\sqsubseteq_{D_{n+1}} = \{(S1, 1264.5), (S1, \top_{D_{n+1}}), (1264.5, \top_{D_{n+1}}), (S2, 5.3), (S2, \top_{D_{n+1}}), (5.3, \top_{D_{n+1}}), (S3, 5.3), (S3, \top_{D_{n+1}}), (S4, 19.1), (S4, \top_{D_{n+1}}), (19.1, \top_{D_{n+1}})\}$ D_{n+1} is put into D, while $E_{D_{n+1}}$ is put into E, AggType is changed as described in the definition. The fact table is not changed.

4.5 Federation Selection

Definition 4.5 (Federation selection operator) Let p be a predicate over the cube $\mathcal{C}$, $\mathcal{F}$ be a federation that is $\{C, Links, X, \mathcal{C}\}$, where $\mathcal{C} = (N, D, E, F, T)$. The Selection operator for a federation , σ_{Fed}, is defined as:

$$\sigma_{Fed(p)}(\mathcal{F}) = \mathcal{F}'$$

where $\mathcal{F}' = (C, Links, X, \mathcal{C}')$, $\mathcal{C}' = (N, D, E, F', T)$,

The only thing changed is the number of tuples. $F' = \{t' | t' \in F \wedge t' \subseteq t'' \wedge t'' \in F_{intermediate} \wedge p(t'') = tt\}$, where, suppose S_p represents the set of expressions of levels in p,

if $S_p = \{\perp_1, \ldots, \perp_l\}$ which means the predicates only contain the bottom levels, then

$$F_{intermediate} = F$$

or if $\{l_x(l_{\perp_x}), \ldots, l_y(l_{\perp_y})\} \subseteq S_p$, $\{l_u[SEM_j]/link_j/xp_j, \ldots, l_v[SEM_k]/link_k/xp_k) \subseteq S_p$

$$F_{intermediate} = F \bowtie R_x \bowtie \ldots \bowtie R_y \bowtie R_{D_{n+j}} \bowtie \ldots \bowtie R_{D_{n+k}}$$

where $\{R_x, \ldots, R_y, R_{D_{n+j}}, \ldots, R_{D_{n+k}}\} \subseteq T$, R_x was built by $\omega_{[\perp_x, l_x]}(\mathcal{F})$, ..., R_y was built by $\omega_{\perp_y, l_y]}(\mathcal{F})$, $R_{D_{n+j}}$ was built by $\delta_{[l_u[SEM_j]link_j/xp_j]}(\mathcal{F})$, ..., $R_{D_{n+k}}$ was built by $\delta_{[l_k[SEM_k]link_k/xp_k]}(\mathcal{F})$.

Example 4.5 For $\sigma_{Fed[Population<30]}(\mathcal{F})$, the decoration values of `Population`, is not present in the fact table, a join is needed. The σ joins the fact table with the temporary table built by $\delta_{[Nation/NLink/Population]}(\mathcal{F})$ in Figure 9. The fact table F before the join is in shown in Figure 6. After the join, we got the table shown below, where the dots represents levels, Part, Order, LineNumber and their values.

Quantity	ExtPrice	Tax	Discount	Supplier	...	Day	**Population**
17	17954	0	0	S1	...	010101	1264.5
36	73638	0	0	S2	...	030201	5.3
28	29983	0	0	S2	...	120302	5.3
26	26374	0	0	S4	...	030201	19.1
2	2388	0	0	S1	...	030201	1264.5

The highlighted column is the newly added column. Then regular SQL selection is performed. Then the levels that are not in F are projected away. The new fact shown in Figure 10 table takes

Quantity	ExtPrice	Tax	Discount	Supplier	Part	Order	LineNumber	Day
36	73638	0	0	S2	grey	33	112	030201
28	29983	0	0	S2	blue	25	123	120302
26	26374	0	0	S4	blue	14	111	030201

Figure 10: The New Fact Table

the place of F.

4.6 Federation Generalized Projection

Definition 4.6 (Generalized projection operator) Let p be a predicate over the cube $\mathcal{C}$, $\mathcal{F}$ be a federation that is $(C, Links, X, \mathcal{C})$, where $\mathcal{C} = (N, D, E, F, T)$. The generalized projection operator for a federation, Π_{Fed}, is defined as:

$$\Pi_{Fed[\perp_p,\ldots,\perp_q,l_s,\ldots,l_t,l_{xp_u},\ldots,l_{xp_w}]<f_x(M_x),\ldots,f_y(M_y)>}(\mathcal{F}) = \mathcal{F}'$$

where $\perp_p \in LS_p, \ldots, \perp_q \in LS_q, l_s \in LS_s, \ldots, l_t \in LS_t, l_{xp_u} \in LS_{D_{n+u}}, \ldots, l_{xp_w} \in LS_{D_{n+w}}, l_s \neq \perp_s, \ldots, l_t \neq \perp_t, \{D_{n+u}, \ldots, D_{n+w}\}$ is a set of dimensions built by decoration operators.

$\{M_x, \ldots, M_y\} \subseteq \{M_1, \ldots, M_m\}$. $\{f_x, \ldots, f_y\}$ are the given aggregate functions for the specified measures, and $\forall f_z \in \{f_x, \ldots, f_y\} \forall D_g \in \{D_p, \ldots, D_q, D_s, \ldots, D_t, D_{n+u}, \ldots, D_{n+w}\}$ $(f_z \in AggType(M_z, D_g))$.

The cube is given by $\mathcal{F}' = (C, Links, X, \mathcal{C}'), \mathcal{C}' = (N, D', E', F', T')$, where $D' = \{D_p, \ldots, D_q, D_s, \ldots, D_t, D_{n+u}, \ldots, D_{n+w}\}$, $E' = \{E_{D_p}, \ldots, E_{D_q}, E_{D_s}, \ldots, E_{D_t}, E_{D_{n+u}}, \ldots, E_{D_{n+w}}\}$, which means unspecified dimensions are rolled up to the top level and projected away. Only the temporary table containing the values required by the federation projection are left, that is, $T' = \{R_t, \ldots, R_t, R_{D_{n+u}}, \ldots, R_{D_{n+w}}\}$, where R_s was built by $\omega_{[\perp_s, L_s]}(\mathcal{F})$,..., R_t was built by $\omega_{[\perp_t, L_t]}(\mathcal{F})$, $R_{D_{n+j}}$ was built by $\delta_{[l_u[SEM_j]link_j/xp_j]}(\mathcal{F})$, ..., $R_{D_{n+k}}$ was built by $\delta_{[l_k[SEM_k]link_k/xp_k]}(\mathcal{F})$.

$$F' = \{(e_p, \ldots, e_q, e_s, \ldots, e_t, v'_u, \ldots, v'_v) | (e_p, \ldots, e_q, e_s, \ldots, e_t, v'_u, \ldots, v'_v) \in {}_{\perp_p,\ldots,\perp_q,l_s,\ldots,l_t}\mathcal{G}_{f_x(M_x),\ldots,f_y(M_y)}(F_{intermediate})\}$$

where

$$F_{intermediate} = F \bowtie R_s \bowtie \ldots \bowtie R_t \bowtie R_{D_{n+u}} \bowtie \ldots \bowtie R_{D_{n+w}}$$

$\mathcal{G}$ is the regular aggregation operation.

Furthermore, if $\exists R_{D_h} \in \{R_{D_{n+u}}, \ldots, R_{D_{n+w}}\} \exists (e_{\perp_h}, e_{xp}) \in R_{D_h} \exists (e_{\perp_h}, e'_{xp}) \in R_{D_h} (e_{xp} \neq e'_{xp})$ then $AggType(M_1, D_h) = c, \ldots, AggType(M_m, D_h) = c$.

if $\exists R_h \in \{R_x, \ldots, R_t\} \exists (e_{\perp_h}, e_h) \in R_h \exists (e_{\perp_h}, e'_h) \in R_h (e_h \neq e'_h)$ then $AggType(M_1, D_h) = c, \ldots, AggType(M_m, D_h) = c$.

Example 4.6 For $\Pi_{Fed[Nation,Supplier]<SUM(Quantity),SUM(ExtPrice)>}(\mathcal{F})$, the query evaluator joins the fact table F in Figure 10 with the table loaded by $\omega_{[Nation,Supplier]}(\mathcal{F})$ shown in Figure 7. Then Pi performs a regular SQL aggregation. The new fact table is shown below.

Quantity	ExtPrice	Supplier	Nation
64	103621	S2	DK
26	26374	S4	UK

4.7 Cube Selection

Definition 4.7 (Cube Selection Operator) Let p be a predicate over the cube $\mathcal{C}$, $\mathcal{F}$ be a federation that is $(C, Links, X, \mathcal{C})$, where $C = (N, D, E, F)$, θ be a predicate over the set of levels $\{l_1, \ldots, l_k\}$ and measures $M_1, \ldots, M_m$. A cube selection is

$$\sigma_{Cube[\theta]}(\mathcal{F}) = \mathcal{F}'$$

where $\mathcal{F} = \{C', Links, X, \mathcal{C}\}$, $C' = \{N, D, E, F'\}$. For more details about cube selection, see [18]

Example 4.7 A cube selection is used to slice the base cube in the OLAP component and can only use predicates referring to dimension levels. The only changes happen on the fact table. After $\sigma_{Cube[Nation='DK']}(\mathcal{F})$, the original fact table in Figure 6 becomes the table shown below.

Quantity	ExtPrice	Tax	Discount	Supplier	Part	Order	LineNumber	Day
36	73638	0	0	S2	grey	33	112	030201
28	29983	0	0	S2	blue	25	123	120302

4.8 Cube Generalized Projection

Definition 4.8 (Cube Generalized Projection Operator) Let p be a predicate over the cube $\mathcal{C}$, $\mathcal{F}$ be a federation that is $(C, Links, X, \mathcal{C})$, where $C = (N, D, E, F)$. A cube generalized projection is defined as:

$$\Pi_{Cube[l_{i_1},\ldots,l_{i_k}]<f_{j_1}(M_{j_1}),\ldots,f_{j_l}(M_{j_l})>}(\mathcal{F}) = \mathcal{F}'$$

where $l_{i_1}, \ldots, l_{i_k}$ levels specifying the aggregation level such that at most one level form each dimension occurs. The measure $\{M_{j_l}, \ldots, M_{j_l}\} \subseteq \{M_1, \ldots, M_m\}$ are kept in the cube and $f_{j_1}, \ldots, f_{j_l}$ are the given aggregate functions for the specified measures.

The resulting federation is $\mathcal{F} = \{C', Links, X, \mathcal{C}\}$, where $C' = \{N, D', E', F'\}$. Dimensions whose levels are specified as parameters of the operator are rolled up to those given levels. All other

dimensions are aggregated to the top level and removed. Each new measure value is calculated by applying the given aggregate function to the corresponding value for all tuples in the fact table containing old bottom values that roll up to the new bottom values. For more details about cube projection, see [18]

Example 4.8 After $\Pi_{Cube[Supplier]<SUM(Quantity),SUM(ExtPrice)>}(\mathcal{F})$, the cube is rolled up to the level, Supplier and aggregated. Therefore, only the dimension, Suppliers, is left where the bottom level is still Supplier.

Quantity	ExtPrice	Tax	Discount	Supplier
19	20342	0	0	S1
64	103621	0	0	S2
26	26374	0	0	S4

4.9 Inlining

Definition 4.9 Let p be a predicate over the cube $\mathcal{C}$, $\mathcal{F}$ be a federation that is $(C, Links, X, \mathcal{C})$, where $\mathcal{C} = (N, D, E, F, T)$. An inlining operator does not change the federation, that is:

$$\iota_{\{[\theta_1,...,\theta_n]\}}(\mathcal{F}, \{\tau_{l_1/Link_1/xp_1}(\mathcal{F}), \ldots, \tau_{l_m/Link_m/xp_m}\}(\mathcal{F})) = \mathcal{F}$$

where $\{\tau_{l_1/Link_1/xp_1}(\mathcal{F}), \ldots, \tau_{l_m/Link_m/xp_m}(\mathcal{F})\}$ is a set of XML-transfers used to load decoration values,θ_i is a predicate with references to level expressions, that is:

$$\theta_i = \begin{cases} l_z[SEM]/Link/xp\ po\ K, \text{ where } K \text{ is a constant value} \\ l_z[SEM]/Link/xp\ po\ l_w, \text{ where } l_w \text{ is a level.} \\ l_z[SEM]/Link/xp\ po\ M, \text{ where } M \text{ is a measure.} \\ l_z[SEM_1]/Link_1/xp_1\ \ po\ \ l_w[SEM_2]/Link_2/xp_2\} \\ l_z[SEM]/Link/xp\ \ IN\ \ (K_1, \ldots, K_n), \text{ where } K_i \text{ is a constant value.} \\ NOT\ \theta_{i1} \\ \theta_{i1}\ bo\ \theta_{i2} \end{cases}$$

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

During evaluation of a query, an inlining operator rewrites the predicates in its parameters and all the other instances of the same predicates in the query. To provide the inlined decoration values, the XML-transfers of an inlining operator are always evaluated first, the rest part of the plan is evaluated after the inlining processes are finished.

After inlining, θ_i is transformed to θ_i' by function $\mathcal{T}(\theta)$, and $\mathcal{T}(\theta_i)$ returns:

1. "$l_z\ \ IN\ \ (t_1, \ldots, t_n)$", if $\theta_i = l_z[SEM]/Link/xp\ \ po\ \ K$, where $t_i \in \{e_z | (e_z, e_{xp}) \in R_{\tau_{l_z[SEM]/Link/xp}} \wedge e_{xp}\ po\ K = true\}$.
2. "$l_z = e_{z1}\ AND\ l_w = e_{xp,1}\ OR\ \ldots\ OR\ l_z = e_{zn}\ AND\ l_w = e_{xp,n}$", if $\theta_i = l_z[SEM]/Link/xp\ po\ l_w$, where $(e_{zi}, e_{xp,i}) \in R_{\tau_{l_z[SEM]/Link/xp}}$.

3. "$l_z = e_{z1}\ AND\ M = e_{xp1}\ OR\ \ldots\ OR\ l_z = e_{zn}\ AND\ M = e_{xpn}$", if $\theta_i = l_z[SEM]/Link/xp\ po\ M$, where $(e_{zi}, e_{xpi}) \in R_{\tau_{l_z[SEM]/Link/xp}}$.

4. " $(l_z = e_{z1}\ AND\ l_w = e_{w1}\ AND\ e_{xp_11} = e_{xp_21}$
   $OR\ \ldots\ OR$
   $l_z = e_{z1}\ AND\ l_w = e_{wn}\ AND\ e_{xp_11} = e_{xp_2n})$
   $OR\ \ldots\ OR$
   $(l_z = e_{zm}\ AND\ l_w = e_{w1}\ AND\ e_{xp_1m} = e_{xp_21}$
   $OR\ \ldots\ OR$
   $l_z = e_{zm}\ AND\ l_w = e_{wn}\ AND\ e_{xp_1m} = e_{xp_2n})$ ",
   if $\theta_i = l_z[SEM_1]/Link_1/xp_1\ \ po\ \ l_w[SEM_2]/Link_2/xp_2$, where $(e_{zi}, e_{xp_1i}) \in R_{\tau_{l_z[SEM]/Link_1/xp_1}}$, $(e_{wi}, e_{xp_2i}) \in R_{\tau_{l_w[SEM]/Link_2/xp_2}}$.

5. $\mathcal{T}(l_z[SEM]/Link/xp = K_1)$ OR ... OR $\mathcal{T}(l_z[SEM]/Link/xp = K_n)$, if $\theta_i = l_z[SEM]/Link/xp\ \ IN\ \ (K_1, \ldots, K_n)$.

6. $\mathcal{T}(\theta_1)\ \ bo\ \ \mathcal{T}(\theta_2)$, if $\theta_i = \theta_{i1}\ \ bo\ \ \theta_{i2}$.

For more details about predicate transformation, see [21].

Example 4.9 Using the example XML document, the predicate "Nation/NLink/Population>100" can be transformed to "Nation='CN"' because only China has the population node whose value is larger than "100".

4.10 Null

Definition 4.10 (Null operator) Let p be a predicate over the cube $\mathcal{C}$, $\mathcal{F}$ be a federation that is $(C, Links, X, \mathcal{C})$, where $\mathcal{C} = (N, D, E, F, T)$. A null operator does nothing, that is:

$$\nu(\mathcal{F}) = \mathcal{F}$$

A null operator can be used to replace a removed operator in the query plan, this kind of null operator has the former operator as its parameter, e.g. a null operator that replaces $\delta_{l_z/Link/xp}(\mathcal{F})$ is $\nu_{\delta_{l_z/Link/xp}}(\mathcal{F})$.

5 Extended Query Semantics and Plan

5.1 Extended Query Semantics

Definition 5.1 (Extended Semantics of an SQL$_{XM}$ query over a federation) In the following:

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

- $\{\perp_p, \ldots, \perp_q\} \subseteq \{\perp_1, \ldots, \perp_d\}$ and $\{l_s, \ldots, l_t\}$ are levels in C such that $\perp_s \sqsubseteq_s l_s, \ldots, \perp_t \sqsubseteq_t l_t$, d is the number of dimensions.
- $l_{z_i}[SEM]/link_i/xp_i$ is a level expression. l_{z_i} is a level in C,$link_i$ is a link from l_{z_i}, and $l_{z_i} \in LS_{z_i}$, $i \in \{u, \ldots, v\}$,$SEM \in \{ANY, ALL, CONCAT\}$
- l_{xp_i} is a decoration level yielded by $l_{z_i}[SEM]/link_i/xp_i$, $i \in \{u, \ldots, v'\}$.
- $f_x, \ldots, f_y$ are distributive aggregation functions.
- $pred_{where}$ represents the predicates in Where clause.
  S_w represents the set of expressions of levels in $pred_{where}$, $\{l_{z_{u_w}}[SEM]/link_{u_w}/xp_{u_w}, \ldots, l_{z_{v_w}}[SEM]/link_{v_w}/xp_{v_w}\} \subseteq S_w$ are the level expressions, $\{l_{s_w}(\perp_{s_w}), \ldots, l_{t_w}(\perp_{s_w})\} \subseteq S_w$ are the roll-up expressions, where $\perp_{s_i} \sqsubseteq_{s_i} l_{s_i}$.
- $pred_{having}$ represents the predicates in Having clause. S_h represents the set of expressions of levels in $pred_{having}$, $\{l_{s_h}(\perp_{s_h}), \ldots, l_{t_h}(\perp_{s_h})\} \subseteq S_h$ are the roll-up expressions, where $\perp_{s_i} \sqsubseteq_{s_i} l_{s_i}$.

Below is a prototypical SQL$_{XM}$ query:

SELECT	$f_x(M_x), \ldots, f_y(M_y), \perp_p, \ldots, \perp_q, l_s, \ldots, l_t,$ $l_{z_u}[SEM]/link_u/xp_u, \ldots, l_{z_v}[SEM]/link_v/xp_v$
FROM	$\mathcal{F}$
WHERE	$pred_{where}$
GROUP BY	$\perp_p, \ldots, \perp_q, l_s, \ldots, l_t,$ $l_{z_u}[SEM]/link_u/xp_u, \ldots, l_{z_v}[SEM]/link_v/xp_v$
HAVING	$pred_{having}$

The physical algebra of a SQL_{XM} query is in Figure 11:

5.2 Extended Query Plan

Figure 12 shows an example of the extended plan tree for the SQL$_{XM}$ query below.

SELECT	SUM(Quantity),SUM(ExtPrice),Nation(Supplier), Supplier
FROM	TData
WHERE	Nation/NLink/Population<50
GROUP BY	Nation(Supplier),Supplier
HAVING	SUM(Quantity)>10000

6 Query Optimizer

The query optimizer is composed of two components: a query *rewriter* and a query *planner*. As shown in Figure 5, the rewriter accepts the initial query plan (to which the query is mapped to) and generates a plan space of possible execution plans which are all equivalent in terms of their final

$$\Pi_{Fed[\perp_p,\ldots,\perp_q,l_s,\ldots,l_t,l_{xp_u},\ldots,l_{xp_v}]<f_x(M_x),\ldots,f_y(M_y)>}(\quad (1)$$

$$\omega_{(\perp_s,l_s)}(\mathcal{F}),\ldots,\omega_{(\perp_t,l_t)}(\mathcal{F}), \quad (2)$$

$$\delta_{[l_{z_u}[SEM]/link_u/xp_u]}(\quad (3)$$

$$\tau_{[l_{z_u}[SEM]/link_u/xp_u]}(\mathcal{F}), \quad (4)$$

$$\omega_{(\perp_{z_u},l_{z_u})}(\mathcal{F})), \quad (5)$$

$$\vdots, \quad (6)$$

$$\delta_{[l_{z_v}[SEM]/link_v/xp_v]}(\quad (7)$$

$$\tau_{(l_{z_v}[SEM]/link_v/xp_v)}(\mathcal{F}) \quad (8)$$

$$\omega_{(\perp_{z_v},l_{z_v})}(\mathcal{F})), \quad (9)$$

$$\sigma_{Fed[pred_{having}]}(\quad (10)$$

$$\omega_{(\perp_{s_h},l_{s_h})}(\mathcal{F}),\ldots,\omega_{(\perp_{t_h},l_{t_h})]}(\mathcal{F}), \quad (11)$$

$$\Pi_{Fed[\perp_p,\ldots,\perp_q,l_s,\ldots,l_t,l_{xp_u},\ldots,l_{xp_v}]<f_x(M_x),\ldots,f_y(M_y)>}(\quad (12)$$

$$\omega_{(\perp_s,l_s)}(\mathcal{F}),\ldots,\omega_{(\perp_t,l_t)}(\mathcal{F}), \quad (13)$$

$$\delta_{[l_{z_u}[SEM]/link_u/xp_u]}(\quad (14)$$

$$\tau_{[l_{z_u}[SEM]/link_u/xp_u]}(\mathcal{F}), \quad (15)$$

$$\omega_{(\perp_{z_u},l_{z_u})}(\mathcal{F})), \quad (16)$$

$$\vdots, \quad (17)$$

$$\delta_{[l_{z_v}[SEM]/link_v/xp_v]}(\quad (18)$$

$$\tau_{(l_{z_v}[SEM]/link_v/xp_v)}(\mathcal{F}) \quad (19)$$

$$\omega_{(\perp_{z_v},l_{z_v})}(\mathcal{F})), \quad (20)$$

$$\sigma_{Fed[pred_{where}]}(\quad (21)$$

$$\omega_{(\perp_{s_w},l_{s_w})}(\mathcal{F}),\ldots,\omega_{(\perp_{t_w},l_{t_w})}(\mathcal{F}), \quad (22)$$

$$\delta_{[l_{z_{u_w}}[SEM]/link_{u_w}/xp_{u_w}]}(\quad (23)$$

$$\tau_{[l_{z_{u_w}}[SEM]/link_{u_w}/xp_{u_w}]}(\mathcal{F}), \quad (24)$$

$$\omega_{[\perp_{z_{u_w}},l_{z_{u_w}}]}(\mathcal{F})), \quad (25)$$

$$\vdots, \quad (26)$$

$$\delta_{[l_{z_{v_w}}[SEM]/link_{v_w}/xp_{v_w}]}(\quad (27)$$

$$\tau_{[l_{z_{v_w}}[SEM]/link_{v_w}/xp_{v_w}]}(\mathcal{F}) \quad (28)$$

$$\omega_{[\perp_{z_{v_w}},l_{z_{v_w}}]}(\mathcal{F})), \quad (29)$$

$$\phi(\quad (30)$$

$$\Pi_{Cube[\perp_{D_1},\ldots,\perp_{D_d},M_1,\ldots,M_m]}(\mathcal{F}))))))))) \quad (31)$$

Figure 11: Semantics of a SQL$_{XM}$ query

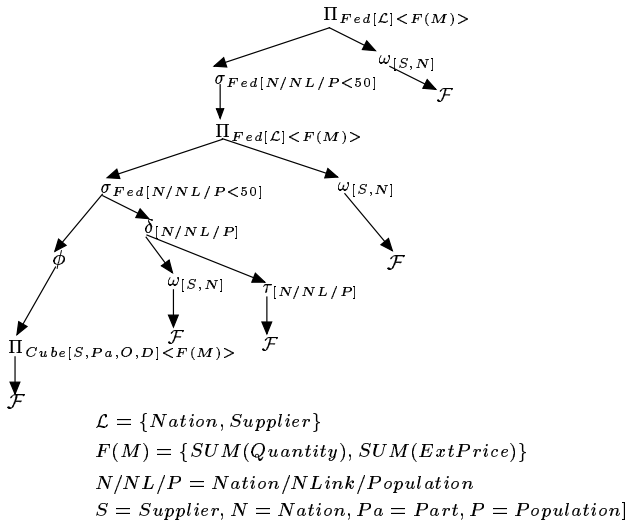


Figure 12: An extended plan tree

output but vary in their cost, i.e., the evaluation time. Equivalent plans are found by *transformation rules* (Section 7). Not all the equivalent plans will be kept in the plan space, instead, some plans will be removed by *pruning techniques* (Section 8), and only plans cheaper than the initial plan will be left. The plan space is passed to the query *planner*, which chooses the plan that needs the least amount of time. The evaluator then follows to process the selected execution plan and produces the result.

The optimizer is based on the Volcano optimizer [11]. The Volcano optimizer optimizes queries in two stages. First, the optimizer generates the entire plan space consisting of logical expressions generated using the initial query plan and set of transformation rules. During the second stage, the search for the best plan is performed. The implementation rules are used to replace operators by algorithms, and the costs of diverse subplans are estimated. The plans from the first stage are logical plans which are then turned into physical access plans in the second stage. Cost estimates and plan selection are done when the operators in logical plans are replaced by implementation algorithms. The rewriter in the OLAP-XML optimizer works in the same way as the first stage of Volcano. Equivalent plans are generated by using transformation rules given the initial query plan. Additionally, *cost-based* and *restriction-based* pruning techniques (Section 8) are integrated to reduce the plan space during the enumeration. As described in Section 4, each operator in a plan is itself physical, moreover, currently, no alternative implementation algorithms are developed. Therefore, the planner in OLAP-XML optimizer does not need to do transformation from logical plans to physical access plans. It just selects the one with the least cost.

6.1 The Rewriter

The rewriter uses transformation rules to generate equivalent plans from the initial plan. In the tree form, a plan is represented by the root operator. Each operator contains the operator type, the parameters (e.g. predicates for the selection) and the pointers to the argument operators. Two equivalent plans yield the same output, that is, the same federation. The plans are held in a list called *plan space*.

Figure 13 and 14 outline simplified pseudo-code for the two main functions for *Rewriter*, *rewrite* and *matchAndApplyRules*. The *rewrite* function first checks if the input *op* represents a plan for which the equivalent plans have already been generated. *planBank* is a global hash-table that stores the equivalent plans for a plan in algebraic expression form. If the equivalent plans already exist, then no more processing is needed and the plans are returned. If the equivalent plans haven not been enumerated but *planBank* has an entry for the input plan, then the function returns NULL. This condition may be satisfied when a plan is rewritten to some form and then written back to the same original form again by applying some rule, e.g. a bidirectional rule. The function returns NULL to avoid falling into an infinite loop. The following condition is satisfied when the input plan has never been rewritten before, the function just adds an entry for the input and continue with the rewriting process. In the following, the *rewriter* first generates the equivalent plans for the input plan's subplans, then generates equivalent plans for the derivatives of the input plan, which are the compositions of the initial plan's root operator and the new subplans. The *rewriter* repeatedly invokes the *matchAndApplyRules* function, which applies a transformation rule to the given operator, choosing from the list of applicable rules that have not so far been applied. A transformation rule is applicable when the root operator, operators below and their parameters all match one side of the rule. In *matchAndApplyRules*, $rule_i$ is a function that determines the applicability and applies the rule if applicable (Section 10.1). The application of a transformation rule may trigger the reconstruction of a plan tree. For example, $\sigma_{Cube[\theta]}(\delta_{Cube[\mathcal{L}]<F(M)>}(\mathcal{F})) \leftrightarrow \delta_{Cube[\mathcal{L}]<F(M)>}(\sigma_{Cube[\theta]}(\mathcal{F}))$ is a rule that switches the two cube operators. Both parts represent a fragment plan in a complete plan and $\mathcal{F}$ is the bottom federation or the output of another operator (Section 7). The *rewrite* function also calls functions, *pruneByCost* and *pruneByRestrictions*, which examine the given plan and use *Branch and Bound* and *restriction-based* pruning techniques to decide whether to drop the plan. *pruneByCost* returns **TRUE** if the argument plan is more expensive than the input plan of the current *rewrite* function. *pruneByRestrictions* returns **TRUE** if the argument plan satisfies some restricted patterns (Section 8). The plan space for the initial plan will be significantly reduced when the subplan spaces are effectively pruned.

Note that the plan space is only pruned just before the function returns, which means, after the loop on Line 18, all equivalent plans are enumerated and only got pruned for the caller of the current *rewrite*. This mechanism ensures the rewriting speed, and at the same time the farsightedness for

generating the least cost plan.

```
List rewrite(Operator op)
1)  {
2)      if planBank has stored the equivalent plans for the logical expression of the plan
        rooted at op
3)          return the list of equivalent plans;
4)      if planBank has an entry for the expression of the plan but has no corresponding
        equivalent plans
5)          return NULL;
6)      if planBank has no entry for the expression of the plan
7)          add the entry;
8)      oplist = NULL;
9)      planlist = NULL;
        /* subOp is the operator below op in the plan tree on the path to the bottom F */
10)     tempList_1 = rewrite(subOp);
11)     merge tempList_1 into opList;
12)     for each operator subOp' in opList;
13)     {
14)         make a copy op' of op;
15)         put subOp' below op' by reference;
16)         tempList_2 = matchAndApplyRules(op');
17)         merge tempList_2 into planList;
18)     }
19)     for each operator op' in planList
20)     {
21)         isPrunedByRestrictions = pruneByRestrictions(op');
22)         isPrunedByCost = pruneByCost(op');
23)         if isPrunedByCost = TRUE and isPrunedByRestrictions = TRUE;
24)             remove op' from planList;
25)     }
26)     return planList;
27)  }
```

Figure 13: Pseudo-code for the Rewriter

```
List matchAndApplyRules(Operator op)
1)  {
2)      planlist = NULL;
3)      for each rule_i in the rule space
4)      {
5)          /*apply the rule if applicable*/
6)          newOp = rule_i(op);
7)          if newOp ≠ NULL
8)          {
9)              tempList_1 = rewrite(newOp);
10)             merge tempList_1 into planList;
11)         }
12)         else
13)             put op into planList;
14)     }
15)     return planList;
16) }
```

Figure 14: Pseudo-code for the Rewriter

6.2 The Planner

The *planner* selects the least cost one. No cost estimation activities are needed for the plans in the plan space, since the *rewriter* has estimated the cost in the plan pruning stage. Figure 15 outlines the pseudo-code for the *planner* function. It iterates through the root operators in the list, and selects the least cost one.

7 Transformation Rules

The transformation rules are given as equivalences that express that two plan fragments in physical algebraic expressions generate the exactly same semantics, i.e. given the same federation, the output federations have the same temporary cube, dimensions, measures, fact table and temporary tables. In the formal presentation, a left-to-right rule (denoted by $\rightarrow$) can only reconstruct the plan fragment on the left into the form on the right side, while each side in a bidirectional rule (denoted by $\leftrightarrow$) can be reconstructed to the algebraic expression on the other side. A plan matches a rule when part of the plan matches the expression on the left side of a left-to-right transformation rule, or the expression on either side of a bidirectional rule. An equivalent plan can then be constructed based

```
Operator planner(List opList)
1)  {
2)      leastCost = COST_UPPERBOUND;
3)      leastCostPlanRoot = NULL;
4)      for each op in opList
5)      {
6)          get the overall cost opCost of the plan;
7)          if leastCost < opCost
8)          {
9)              leastCost = opCost;
10)             leastCostPlanRoot = op;
11)         }
12)     }
13)     return leastCostPlanRoot;
14)  }
```

Figure 15: Pseudo-code for the Planner

on the expression on the other side of the matching rule.

In the following transformation rules,

1. Let R be a table, $R \in T$. The set of attributes of R, is denoted as Ω_R.
2. Let $\mathcal{L}$ be a set of levels from different dimensions.
3. Let $F(M) = \{f_1(M_1), \ldots, f_s(M_s)\}$ be a set of aggregation functions applied to measures.
4. Υ be a set of *secondary argument* operators. A secondary argument operator is an operator that does not change the federation's fact table, e.g. a decoration operator is a secondary argument operator because decoration builds temporary dimensions and does not have a direct impact on the fact table. In Figure 16, δ is a secondary argument operator for σ_{Fed}. The operators of δ are all secondary argument operators.

   In the arguments of an operator, $\mathcal{F}$ is taken as a non-secondary argument operator, e.g. in the arguments of $\sigma_{[\theta]}(\mathcal{F}, \Upsilon)$, $\mathcal{F}$ represents an operator like, Π_{Fed} or ϕ.
5. We use Ω to denote the set of all operators, i.e. $\Omega = \{\phi, \omega, \tau, \delta, \sigma_{Fed}, \Pi_{Fed}, \sigma_{Cube}, \Pi_{Cube}, \iota, \nu\}$.
6. An operator symbol without subscripts is an operator with whatever parameters.
7. *SubOps*(Op) is a function that returns the secondary argument operators of operator Op, where $Op \in \Omega$.

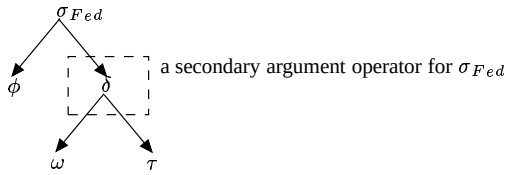


Figure 16: Secondary argument operators in a subplan

8. $Op_{[params_1]}(\mathcal{F}) = Op_{[params_2]}(\mathcal{F})$ if parameters, $params_1 = params_2 \wedge Op \in \Omega$, e.g. $\sigma_{Cube[\theta_1]}(\mathcal{F}) = \sigma_{Cube[\theta_2]}(\mathcal{F})$. In short, we write $Op_1 = Op_2$ to denote this equivalence.

9. The *union rule* is used when we make union of sets of operators, that is, let C, A and B be three sets of operators, $C = A \cup B$, $\forall c \in C$,

$$c = \begin{cases} a & if\ a \in A \wedge b \in B \wedge a = b \wedge \left|SubOps(a)\right| > \left|SubOps(b)\right| \\ b & if\ a \in A \wedge b \in B \wedge a = b \wedge \left|SubOps(a)\right| < \left|SubOps(b)\right| \\ e & if\ e \in A \wedge e \notin B \\ f & if\ f \notin A \wedge f \in B \\ g & if\ \nu_g \in A \wedge g \in B \\ h & if\ g \in A \wedge \nu_g \in B \end{cases}$$

   where $a, b \in \Omega$, while $e, f, g, h \in \Omega \backslash \{\nu\}$.

   Similarly, the *intersection rule* is, when $C = A \cap B$, $\forall c \in C$,

$$c = \begin{cases} a & if\ a \in A \wedge b \in B \wedge a = b \wedge \left|SubOps(a)\right| > \left|SubOps(b)\right| \\ b & if\ a \in A \wedge b \in B \wedge a = b \wedge \left|SubOps(a)\right| < \left|SubOps(b)\right| \\ g & if\ \nu_g \in A \wedge g \in B \\ h & if\ g \in A \wedge \nu_g \in B \end{cases}$$

Rule 7.1 (Commutativity of Federation Generalized Projection and Federation Selection) The following rule holds:

$$\Pi_{Fed[\mathcal{L}]<F(M)>}(\sigma_{Fed[\theta]}(\mathcal{F}, \Upsilon_{\sigma_{Fed}}), \Upsilon_{\Pi_{Fed}}) \leftrightarrow \sigma_{Fed[\theta]}(\Pi_{Fed[\mathcal{L}]<F(M)>}(\mathcal{F}, \Upsilon'_{\Pi_{Fed}}), \Upsilon'_{\sigma_{Fed}})$$

First of all, the federation selection does not refer to measures. The right to left direction of the rule is always true, the other direction holds if $\forall l_i \in RLevels(\theta) \exists \perp_i (\perp_i \sqsubseteq_i l_i \wedge \perp_i \in \mathcal{L})$,

RLevels(θ) is a function that returns all the referenced dimension levels in θ, that is:

$$RLevels(\theta) = \begin{cases} \{l_z\} & \text{if } \theta = l_z \ po \ K \\ \{l_z, l_w\} & \text{if } \theta = l_z \ po \ l_w \\ \{l_z\} & \text{if } \theta = l_z \ po \ M \\ \{l_z\} & \text{if } \theta = l_z[SEM]/Link/l_{xp} \ po \ K \\ \{l_z, l_w\} & \text{if } \theta = l_z[SEM]/Link/l_{xp} \ po \ l_w \\ \{l_z\} & \text{if } \theta = l_z[SEM]/Link/l_{xp} \ po \ M \\ \{l_z, l_w\} & \text{if } \theta = l_z[SEM]/Link/l_{xp_1} \ po \ l_w \\ RLevels(\theta_1) & \text{if } \theta = NOT \ \theta_1 \\ RLevels(\theta_1) \cup RLevels(\theta_2) & \text{if } \theta = \theta_1 \ AND \ \theta_2 \\ \phi & \text{Otherwise} \end{cases}$$

where, the binary operator *bo* is either AND or OR, the predicate operator po is one of : $=, <, >$, $<>, >=, <=$ and LIKE, K is a constant value, and M is a measure.

For the left to right direction, $\Upsilon'_{\Pi_{Fed}} = \Upsilon_{\Pi_{Fed}} \cup \Upsilon_{temp}$, $\Upsilon'_{\sigma_{Fed}} = \Upsilon_{\sigma_{Fed}} \backslash \Upsilon_{temp}$, where, $\Upsilon_{temp} = \big\{\{\omega_{[\perp_i, l_i]}(\mathcal{F}), \delta_{[l_j/Link/xp]}(\mathcal{F})\} | l_i \in RLevels(\theta) \wedge \perp_i \neq l_i \wedge l_i \in \mathcal{L} \wedge l_i \rightsquigarrow \omega_{[\perp_i, l_i]}(\mathcal{F}) \wedge l_j/Link/xp \in RLExprs(\theta) \wedge l_j/Link/xp \in \mathcal{L} \wedge l_j/Link/xp \rightsquigarrow \delta_{[l_j/Link/xp]}(\mathcal{F})\big\}$, *RLExprs*($\theta$) is the function that returns the referenced level expressions in the predicates, that is:

$$RLExprs(\theta) = \begin{cases} \{l_z[SEM]/Link/l_{xp}\} & \text{if } \theta = l_z[SEM]/Link/l_{xp} \ po \ K \\ \{l_z[SEM]/Link/l_{xp}\} & \text{if } \theta = l_z[SEM]/Link/l_{xp} \ po \ l_w \\ \{l_z[SEM]/Link/l_{xp}\} & \text{if } \theta = l_z[SEM]/Link/l_{xp} \ po \ M \\ \{l_z[SEM_1]/Link_1/l_{xp_1}, \\ \quad l_w[SEM_2]/Link_2/l_{xp_2}\} & \text{if } \theta = l_z[SEM_1]/Link_1/l_{xp_1} \ po \\ & \quad l_w[SEM_2]/Link_2/l_{xp_2} \\ RLExprs(\theta_1) & \text{if } \theta = NOT \ \theta_1 \\ RLExprs(\theta_1) \cup RLExprs(\theta_2) & \text{if } \theta = \theta_1 \ AND \ \theta_2 \\ \{\epsilon\} & \text{Otherwise} \end{cases}$$

For the right to left direction, $\Upsilon_{\Pi_{Fed}} = \Upsilon'_{\Pi_{Fed}} \backslash \Upsilon_{temp}$, $\Upsilon_{\sigma_{Fed}} = \Upsilon'_{\sigma_{Fed}} \cup \Upsilon_{temp}$.

Reasoning: The federation selection can slice the temporary cube when the required bottom levels are present in the fact table so that joins can be performed. The federation GP does not change the dimension values for the levels in the GROUP BY clause, exactly the same facts will be selected.

To illustrate more clearly in this example and the following ones, we give the partial plans in the form of trees.

Example 7.1 In Figure 17, the left partial plan is rewritten to the right one. $\sigma_{Fed[C='Mike' \ AND \ N='DK']}$ has two secondary argument operators, one of which, $\omega_{S,N}$ is also the child of $\Pi_{Fed[\mathcal{L}]<F(M)>}$. In the transformed plan, the null operator replaces the duplicate which means the selection will use the dimension values loaded by the bottom $\omega_{[N,S]}$.

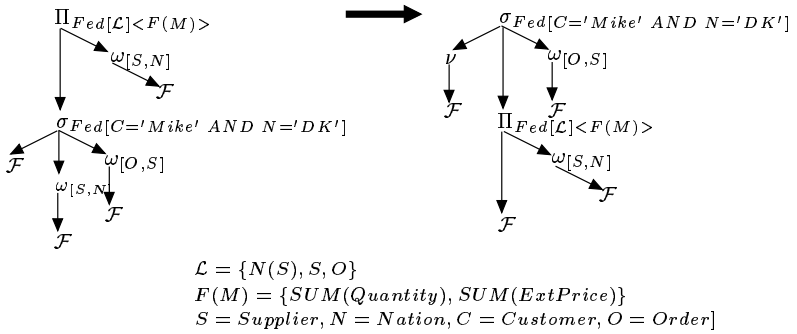


Figure 17: Community of Federation Generalized Projection and Selection

The two functions, $RLevels(\theta)$ and $RLExprs(\theta)$ also stand in the following descriptions.

Rule 7.2 (Pushing Federation Generalized Projection Below Federation Selection) The following rule holds:

$$\Pi_{Fed[\mathcal{L}]<F(M)>}(\sigma_{Fed[\theta]}(\mathcal{F}, \Upsilon_{\sigma_{Fed}}), \Upsilon_{\Pi_{Fed}}) \rightarrow$$
$$\Pi_{Fed[\mathcal{L}]}(\sigma_{Fed[\theta]}(\Pi_{Fed[\mathcal{L}']<F(M)>}(\mathcal{F}, \Upsilon'_{\Pi_{Fed}}), \Upsilon'_{\sigma_{Fed}}), \{\nu, \ldots, \nu_n\})$$

if $\exists l_i \in RLevels(\theta) \exists \perp_i (\perp_i \sqsubseteq_i l_i \wedge \perp_i \notin \mathcal{L})$, and the federation selection does not refer to measures.

In the rule, $\mathcal{L}' = \mathcal{L} \cup \{\perp_i | l_i \in RLevels(\theta) \wedge \perp_i \sqsubseteq_i l_i\}$, $\Upsilon'_{\Pi_{Fed}} = \Upsilon_{\Pi_{Fed}} \cup \Upsilon_{temp}$, $\Upsilon'_{\sigma_{Fed}} = \Upsilon_{\sigma_{Fed}} \backslash \Upsilon_{temp}$, where, $\Upsilon_{temp} = \big\{\{\omega_{[\perp_i, l_i]}(\mathcal{F}), \delta_{[l_j/Link/xp]}(\mathcal{F})\} | l_i \in RLevels(\theta) \wedge \perp_i \neq l_i \wedge l_i \in \mathcal{L} \wedge l_i \rightsquigarrow \omega_{[\perp_i, l_i]}(\mathcal{F}) \wedge l_j/Link/xp \in RLExprs(\theta) \wedge l_j/Link/xp \in \mathcal{L} \wedge l_j/Link/xp \rightsquigarrow \delta_{[l_j/Link/xp]}(\mathcal{F})\big\}$, n is the number of secondary arguments of the federation projection in the header.

Reasoning: The federation selection needs the bottom levels to connect other temporary dimension tables. The new federation GP has the levels not present in the old one, which enables the selection can be performed successfully. Similar to Rule 7.1, same facts will be selected in both case.

Example 7.2 In Figure 18, a new federation generalized projection (GP) with all the bottom levels required by the original two operators is created below the federation selection. The secondary operators of the old projection are moved to the new projection. The common secondary operator, $\omega_{[S,N]}$ is also moved down and replaced by ν.

Rule 7.3 (Pushing Federation Generalized Projection Below Fact-Transfer) The following rule holds:

$$\Pi_{Fed[\mathcal{L}]<F(M)>}(\phi(\mathcal{F}), \Upsilon) \rightarrow \Pi_{Fed[\mathcal{L}]}(\phi(\Pi_{Cube[\mathcal{L}']<F(M)>}(\mathcal{F})), \Upsilon)$$

where $\mathcal{L}' = \{l_i | l_i \in \mathcal{L} \wedge l_i = \perp_i\}$.

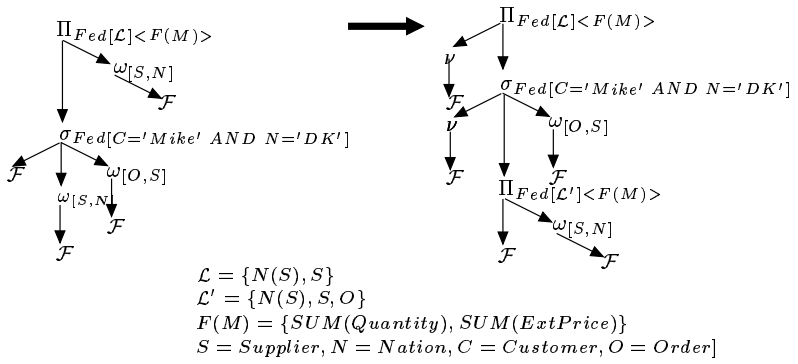


Figure 18: Pushing Federation Generalized Projection below Selection

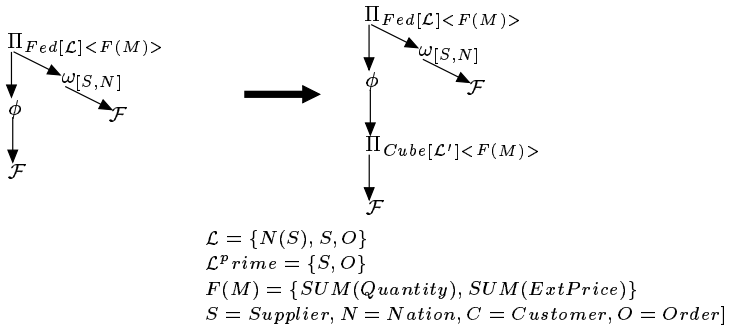


Figure 19: Pushing Federation Generalized Projection below Fact Transfer

A federation GP can produce a cube projection with the same aggregate functions and dimension levels.

Reasoning: The cube projection aggregates measures to a given level and at the same time removes dimensions and measures from a cube. The fact transfer then transfers processed facts of aggregated measures and required levels other than the whole fact table. Federation operators then no longer need to perform aggregate and roll-ups in the temporary database for a not yet decorated cube. Therefore, transfering and processing time can be reduced by adding cube operators.

Example 7.3 In Figure 19, a cube projection is generated below the fact transfer in the right plan.

Rule 7.4 (Cascade of Federation Generalized Projections) Let $\mathcal{L}$ and $\mathcal{L}'$ be sets of levels that $\mathcal{L} \subseteq \mathcal{L}'$ and let $F(M)$ and $F(M)'$ be aggregate functions applied to measures such that $F(M) \subseteq$

$F(M)'$, Then the following holds:

$$\Pi_{Fed[\mathcal{L}]<F(M)>}(\Pi_{Fed[\mathcal{L}']<F(M)'>}(\mathcal{F}, \Upsilon_{\Pi_{Fed[\mathcal{L}']}}), \Upsilon_{\Pi_{Fed[\mathcal{L}]}}) \rightarrow \Pi_{Fed[\mathcal{L}]<F(M)>}(\mathcal{F}, \Upsilon_{\Pi_{Fed[\mathcal{L}]}} \cap \Upsilon_{\Pi_{Fed[\mathcal{L}']}})$$

The lower federation GP can be removed when the upper one rolls up the temporary cube to a higher level. The same secondary child operators are kept, while the others are removed since they will not present in the temporary cube after the federation GP, according to the definition.

Reasoning: This holds because aggregate functions are assumed to be distributive.

Rule 7.5 (Cascade of Federation Selections) The following holds:

$$\sigma_{Fed[\theta_1 \wedge \theta_2]}(\mathcal{F}, \Upsilon) \leftrightarrow \sigma_{Fed[\theta_1]}(\sigma_{Fed[\theta_2]}(\mathcal{F}, \Upsilon_{\sigma_{Fed[\theta_1]}}), \Upsilon_{\sigma_{Fed[\theta_2]}})$$

For the left to right direction,
$\Upsilon_{\sigma_{Fed[\theta_1]}} = \{\omega_{[\perp_z, l_z]}(\mathcal{F}) | l_z \neq \perp_z \wedge l_z \in RLevels(\theta_1) \wedge l_z(\perp_z) \rightsquigarrow \omega_{[\perp_z, l_z]}(\mathcal{F}) \wedge \omega_{[\perp_z, l_z]}(\mathcal{F}) \in \Upsilon\} \cup$
$\{\delta_{[l_z/Link/xp]}(\mathcal{F}) | l_z/Link/xp \in RLExprs(\theta_1) \wedge l_z/Link/xp \rightsquigarrow \delta_{[l_z/Link/xp]}(\mathcal{F}) \wedge \delta_{[l_z/Link/xp]}(\mathcal{F}) \in \Upsilon\}$,
$\Upsilon_{\sigma_{Fed[\theta_2]}} = \{\omega_{[\perp_z, l_z]}(\mathcal{F}) | l_z \neq \perp_z \wedge l_z \in RLevels(\theta_2) \backslash (RLevels(\theta_1) \cap RLevels(\theta_2)) \wedge l_z(\perp_z) \rightsquigarrow \omega_{[\perp_z, l_z]}(\mathcal{F}) \wedge \omega_{[\perp_z, l_z]}(\mathcal{F}) \in \Upsilon\} \cup$
$\{\delta_{[l_z/Link/xp]}(\mathcal{F}) | l_z/Link/xp \in RLExprs(\theta_2) \backslash (RLExprs(\theta_2) \cap RLExprs(\theta_1)) \wedge l_z/Link/xp \rightsquigarrow \delta_{[l_z/Link/xp]}(\mathcal{F}) \wedge \delta_{[l_z/Link/xp]}(\mathcal{F}) \in \Upsilon\}$,

A federation can be split into two federation selections, each of which has its own secondary argument operators, while the inner federation selection has the common operators to load the temporary tables for both selections.

For the right to left direction, $\Upsilon = \Upsilon_{\sigma_{Fed[\theta_1]}} \cup \Upsilon_{\sigma_{Fed[\theta_2]}}$

Two federation selections can be combined into one, the new selection has the union of the secondary argument operators of both selections.

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

Example 7.4 In Figure 20, two federation selections are created for "$C = M$" and "$M \ LIKE \ 'DK\%'$". The bottom selection has both of the dimension transfers to load the dimension values for "Manufacturer" and "Customer". The top selection has no dimension-transfers because the temporary table for values of "Manufacturer" is already loaded.

Rule 7.6 (Commutativity of Federation Selections)

$$\sigma_{Fed[\theta_1]}(\sigma_{Fed[\theta_2]}(\mathcal{F}, \Upsilon_{\sigma_{Fed[\theta_2]}}), \Upsilon_{\sigma_{Fed[\theta_1]}}) \rightarrow \sigma_{Fed[\theta_2]}(\sigma_{Fed[\theta_1]}(\mathcal{F}, \Upsilon'_{\sigma_{Fed[\theta_1]}}), \Upsilon'_{\sigma_{Fed[\theta_2]}})$$

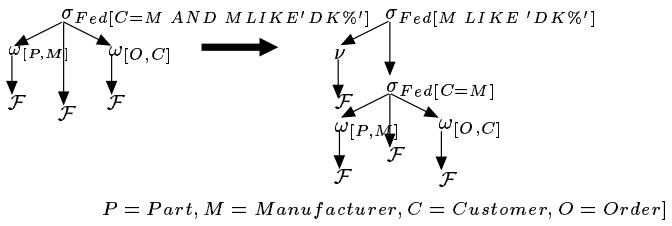


Figure 20: Splitting Federation Selection

where
$\Upsilon'_{\sigma_{Fed[\theta_2]}} = \Upsilon_{\sigma_{Fed[\theta_1]}} \backslash ComnOps$, $\Upsilon'_{\sigma_{Fed[\theta_1]}} = \Upsilon_{\sigma_{Fed[\theta_1]}} \cup ComnOps$,
$ComnOps = AllSubOps(\Upsilon_{\sigma_{Fed[\theta_1]}}) \cap AllSubOps(\Upsilon_{\sigma_{Fed[\theta_2]}})$. Two federation selections can be switched, but the duplicate operators should be removed from the children of the new top selection, because the same operators below will build or load the required temporary tables.

Reasoning: Follows from Rule 7.5 and commutativity of conjunctions.

Rule 7.7 (Pushing Federation Selection below Fact-Transfer) The following rule holds:

$$\sigma_{Fed[\theta]}(\phi(\mathcal{F}), \Upsilon) \rightarrow \nu_{[\sigma_{Fed[\theta]}]}(\phi(\sigma_{Cube[\theta]}(\mathcal{F})), \Upsilon)$$

if $RLExprs(\theta) = \phi$.

A federation selection without references to level expressions can be pushed below the fact-transfer and become a cube selection, while a new null operator with the federation selection's secondary operators is created above the fact-transfer.

Reasoning: A selection with only references to dimension levels can be performed either in temporary or OLAP component.

Example 7.5 In Figure 21, a new cube selection is created below the fact-transfer, the dimension tables are still loaded by the two dimensions which becomes the children of a new null operator.

Rule 7.8 (Combination of Federation Generalized Projection and Null) The following rule holds:

$$\Pi_{Fed[\mathcal{L}]<F(M)>}(\nu(\mathcal{F}, \Upsilon_\nu), \Upsilon_{\Pi)Fed}) \rightarrow \Pi_{Fed[\mathcal{L}]<F(M)>}(\mathcal{F}, \Upsilon'_{\Pi_{Fed}})$$

where $\Upsilon'_{\Pi_{Fed}} = \Upsilon_{\Pi_{Fed}} \cup \{\omega_{[\perp_z, l_z]}(\mathcal{F}) | l_z \neq \perp_z \wedge l_z \in \mathcal{L} \wedge l_z(\perp_z) \rightsquigarrow \omega_{[\perp_z, l_z]}(\mathcal{F}) \wedge \omega_{[\perp_z, l_z]}(\mathcal{F}) \in \Upsilon_\nu\} \cup \{\delta_{[l_z/Link/xp]}(\mathcal{F}) | l_z/Link/xp \in \mathcal{L} \wedge l_z/Link/xp \rightsquigarrow \delta_{[l_z/Link/xp]}(\mathcal{F}) \wedge \delta_{[l_z/Link/xp]}(\mathcal{F}) \in \Upsilon_\nu\}$

A null operator can be combined with a federation GP. The common secondary argument operators are kept, and the rest are removed.

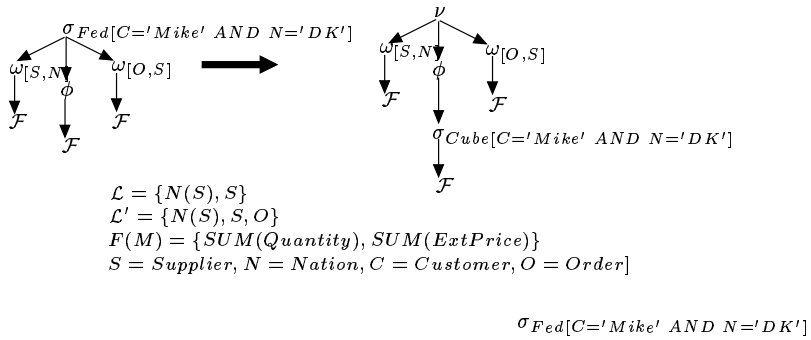


Figure 21: Pushing Federation Selection below Fact-Transfer

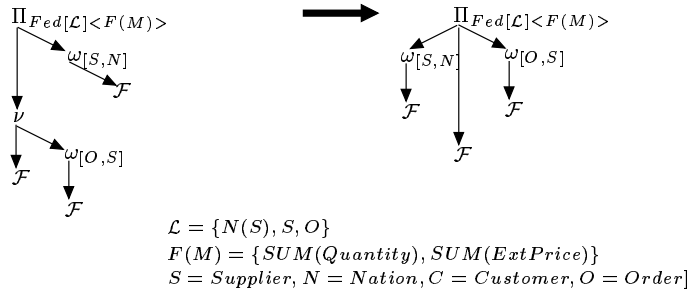


Figure 22: Combination of Federation Generalized Projection and Null Operator

Reasoning: A null operator does nothing but transfer the resulting federation produced by the lower operators. According to the semantics of federation GP, only dimensions and temporary tables that take part in the projection are kept in the resulting federation, therefore, the lower operators that produce useful results are kept.

Example 7.6 In Figure 22, a null operator is combined with the upper federation GP, $\omega)[S, N]$ becomes the child of federation GP.

Rule 7.9 (Combination of Federation Selection and Null) The following rule holds:

$$\sigma_{Fed[\theta]<F(M)>}(\nu(\mathcal{F}, \Upsilon_\nu), \Upsilon_{\sigma_{Fed}}) \rightarrow \sigma_{Fed[\theta]<F(M)>}(\mathcal{F}, \Upsilon_{\sigma_{Fed}} \cup \Upsilon_\nu)$$

A null operator can be combined with a federation GP which inherits all the secondary argument operators of the null operator.

Reasoning: A null operator does nothing but transfers the resulting federation produced by the lower operators. Therefore, we add the lower operators to the federation GP so that to keep the semantics.

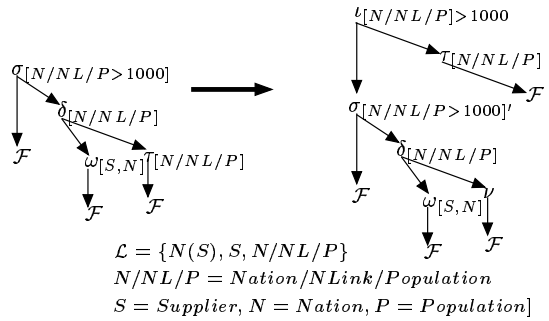


Figure 23: Inlining Decoration Data in Federation Selection

Rule 7.10 (Inlining Decoration Data in Federation Selection) The following rule holds:

$$\sigma_{Fed[\theta]}(\mathcal{F}, \Upsilon) \rightarrow \iota_o(\sigma_{\theta'}(\mathcal{F}, \Upsilon'_1), \Upsilon'_2)$$

where $o = \{\theta'\}$, $\Upsilon'_1 = \{\tau_{l_i/Link/xp}(\mathcal{F}) | \forall l/Link/xp \in RLExprs(\theta)(l/Link/xp \rightsquigarrow \tau_{l_i/Link/xp})\}$, $\Upsilon'_2 = AllSubOps_\Upsilon \backslash \{\tau_{l_i/Link/xp}(\mathcal{F})\}$, θ' is the same as θ but with an *inlinedMark* which tells the query evaluator to transform this predicate into a new one with references only to dimension levels. A predicate with such a mark no longer has references to level expressions. That is, $RLExprs(\theta') = \phi$.

Reasoning: The selection with the new predicate will select the exactly same facts as the old one.

Example 7.7 In Figure 23, an inlining operator is created and the predicate is marked and will be transformed during query evaluation.

Rule 7.11 (Commutativity of Federation Generalized Projection and Inlining) The following rule holds:

$$\Pi_{Fed[\mathcal{L}]<F(M)>}(\iota_o(\mathcal{F}, \Upsilon_\iota), \Upsilon_{\Pi_{Fed}}) \rightarrow \iota_o(\Pi_{Fed[\mathcal{L}]<F(M)>}(\mathcal{F}, \Upsilon'_{\Pi_{Fed}}), \Upsilon_\iota)$$

where $\Upsilon'_{\Pi_{Fed}} = AllSubOps_{\Upsilon_{\Pi_{Fed}}} \backslash \Upsilon_\iota$,
$AllSubOps_{\Upsilon_{\Pi_{Fed}}} = \Upsilon_{\Pi_{Fed}} \cup \{Op' | Op' \in SubOps(Op) \wedge Op \in \Upsilon_{\Pi_{Fed}}\}$.

A federation GP can be pushed below an inlining operator. If they have common child operators, then the duplicates are removed from the federation GP's children.

$$\iota(\Pi_{Fed[\mathcal{L}]<F(M)>}(\mathcal{F}, \Upsilon_{\Pi_{Fed}}), \Upsilon_\iota) \rightarrow \Pi_{Fed[\mathcal{L}]<F(M)>}(\iota(\mathcal{F}, \Upsilon_\iota), \Upsilon'_{\Pi_{Fed}})$$

where $\Upsilon'_{\Pi_{Fed}}$ is defined as above. An inlining operator can be pushed below a federation GP. If they have common child operators, then the duplicates are removed from the federation GP's children.

Reasoning: An inlining operator does not change the fact table and the temporary dimension tables that the federation GP will probably operating on, thus it can be pushed down or moved up. An inlining operator always performs the secondary arguments first and the temporary table yielded by the XML-transfer is always loaded before the operators on the other sub-branch of the inlining operator are executed. Thus the duplicate child operators are only kept for the inlining.

Rule 7.12 (Commutativity of Federation Selection and Inlining) The following rule holds:

$$\sigma_{Fed[\theta_1]}(\iota_o(\mathcal{F}, \Upsilon_\iota), \Upsilon_{\sigma_{Fed}}) \rightarrow \iota_o(\sigma_{Fed[\theta_1]}(\mathcal{F}, \Upsilon'_{\sigma_{Fed}}), \Upsilon_\iota)$$

if $\forall\theta \in o(\theta_1 \neq \theta)$, where $\Upsilon'_{\sigma_{Fed}} = AllSubOps_{\Upsilon_{\sigma_{Fed}}} \backslash \Upsilon_\iota$,
$AllSubOps_{\Upsilon_{\Pi_{Fed}}} = \Upsilon_{\Pi_{Fed}} \cup \{Op' | Op' \in SubOps(Op) \wedge Op \in \Upsilon_{\Pi_{Fed}}\}$.

A federation selection can be pushed below an inlining operator. If they have common child operators, then the duplicates are removed from the federation selection's children.

$$\iota_{[\theta_1]}(\sigma_{Fed[\theta_2]}(\mathcal{F}, \Upsilon_{\sigma_{Fed}}), \Upsilon_\iota) \rightarrow \sigma_{Fed[\theta_2]}(\iota_{[\theta_1]}(\mathcal{F}, \Upsilon_\iota), \Upsilon'_{\sigma_{Fed}})$$

where $\Upsilon'_{\sigma_{Fed}}$ is defined as above. An inlining operator can be pushed below a federation selection. The duplicate child operators are removed from the federation selection's children.

Reasoning: Same as for Rule 7.11.

Rule 7.13 (Cascade of Inlining Operators) The following rule holds:

$$\iota_{o_1}(\iota_{o_2}(\mathcal{F}, \Upsilon_{\iota_{o_2}}), \Upsilon_{\iota_{o_1}}) \rightarrow \iota_o(\mathcal{F}, \Upsilon_{\iota_o})$$

where, $o = o_1 \cup o_2$, $\Upsilon_{\iota_o} = \Upsilon_{\iota_{o_1}} \cup \Upsilon_{\iota_{o_2}}$.

Two inlining operators can be combined into one.

Reasoning: According to the definition, the new predicate strings can be generated by one inlining operator.

Rule 7.14 (Cascade of Cube Projections) The following rule holds:

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

if $\forall l' \in \mathcal{L}' \exists l \in \mathcal{L}(l' \sqsubseteq l) \wedge F(M) \subseteq F(M)'$

Reasoning: This holds because besides the aggregation functions that are assumed to be distributive, the cube projections only have bottom levels in their parameters.

Rule 7.15 (Commutativity of Cube Selection and Cube Projection) The following rule holds if $\forall l \in RLevels(\theta)(l \in \mathcal{L})$ and θ does not refer to measures:

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

Reasoning: The cube projection does not change the values of the dimension levels, and the cube selection does not change the cube schema, thus the projection can be pushed below the cube selection.

Rule 7.16 (Cascade of Cube Selections) Let θ_1 and θ_2 be predicates, Then the following holds:

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

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

Rule 7.17 (Commutativity of Cube Selections)

$$\sigma_{Cube[\theta_1]}(\sigma_{Cube[\theta_2]}(\mathcal{F})) \rightarrow \sigma_{Cube[\theta_2]}(\sigma_{Cube[\theta_1]}(\mathcal{F}))$$

Two cube selections can be switched.

Reasoning: Follows from Rule 7.16 and commutativity of conjunction.

Rule 7.18 (Elimination of Duplicate Secondary Argument Operators) Let Op be an operator, the following rule holds:

$$Op(\mathcal{F}, \Upsilon) \rightarrow Op(\mathcal{F}, \Upsilon')$$

where $\Upsilon' = \Upsilon \backslash (\Upsilon \cap \{Op' | Op' \in SubOps(Op'') \wedge Op'' \in \Upsilon\})$

Duplicate operators should be removed from the set of operators including the operator's secondary argument operators and the child operators of the secondary argument operators.

Reasoning: Duplicate operators yields the exactly same result, therefore the later one is redundant.

Example 7.8 In Figure 24, the $\omega_{Supplier,Nation}(\mathcal{F})$ which is a child of the federation GP, is replaced with a null operator. The $\omega_{Supplier,Nation}(\mathcal{F})$ for the decoration is kept, which loads the dimension values required by the federation GP and the decoration.

8 Plan Space Pruning

Pruning techniques are used to reduce the plan space and therefore increases the optimization speed. A naive way to generate the plan space enumerates all the equivalent plans that the given initial plan could be rewritten to. As the rewriting of a plan starts with the rule matching and applying process for its partial plans. Generated equivalent plans are also taken as the input of the rewriting to generate the other equivalent potential plans. This process is recursively called on all the partial

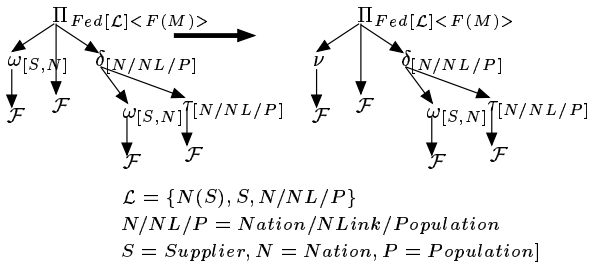


Figure 24: Elimination of Duplicate Secondary Argument Operators

queries until it reaches the basic element of the plans. Thus, the naive approach makes the plan space grow larger and larger significantly as more and more equivalent plans are found.

The plan space greatly affects the optimization time. To speed up, several search strategies are introduced. The idea is that if we can use these strategies to reduce the equivalent plans enumerated for a partial plan, then the total number of plans in the plan space will be reduced. *Branch-and-bound* (B&B) techniques can be used to reduce the amount of search needed to find the optimal solution. B&B is based on a common-sense principle: do not keep trying a path that you already know is worse than the best answer. The other strategy is *restriction-based pruning* [15]. The idea is that a plan will be removed if it matches some special patterns. Observation experiences have shown that the plans that satisfy the restrictions are sure to be more expensive. Moreover, the *restriction-based* pruning is much more cheaper than the other pruning technique, and therefore, it is applied first.

This section outlines the details of these two pruning techniques.

8.1 Branch and Bound

Branch and Bound is by far the most widely used tool for solving complex optimization problems[4]. When applied in our case of query optimization, the pruning idea is to reduce the search by excluding the generated equivalent plans which cannot give less execution time than the original plan, and therefore only plans with less estimated query evaluation time will be kept.

The B&B algorithm adapted in our case is defined below:

Definition 8.1 (Branch and Bound) is an algorithm which uses the cost of an existing plan as an upper bound, and can stop developing a partial plan if its cost exceeds the upper bound.

The algorithm is applied every time a plan space is generated for a given initial plan.

```
1) use the cost of the initial plan as an upper bound K;
2) evaluate the cost of plans in the plan space;
3) for each plan i in plan space
4)         if the cost K' is larger than K
6)                 remove plan i from plan space;
```

8.2 Restriction-based Pruning

The raw plan space contains plans that are fully transformed by all the matched rules, and plans only partially transformed. In other word, the raw plan space records every plan during the rewriting process. Some of the plans will definitely not be selected later as the execution plan by the planner, and some can be found to be expensive even without cost estimate. All these plans need to be pruned during the rewriting process before the plan space is handed to the planner, thus, the algorithm, restriction-based pruning, is introduced.

Definition 8.2 (Restriction-based Pruning) is an algorithm that filters out the plans satisfying some special patterns for an individual query.

Currently, the patterns we have found are:

- Matching the rules that remove redundant operators but not applied.
- Containing operators that produce the same temporary tables.

Algorithm:

```
1) for each plan i in the plan space
2)         for each pattern
3)                 go through the nodes of the plan and see if it matches
the pattern;
4)                 if match
5)                         remove plan i from plan space;
```

Example 8.1 When Rule 7.14 is matched but not applied, the plans with the two adjacent operators shown in Figure 25 will be removed from the plan space for the initial plan.

Example 8.2 Plans with duplicate decorations, dimension transfers or XML transfers will generate redundant temporary tables. Thus these plans will be pruned.

9 Query Cost Estimation

This section describes the statistics information and the cost formulas used for cost estimation. Querying cost estimation is needed to measure the evaluation time for query plans.

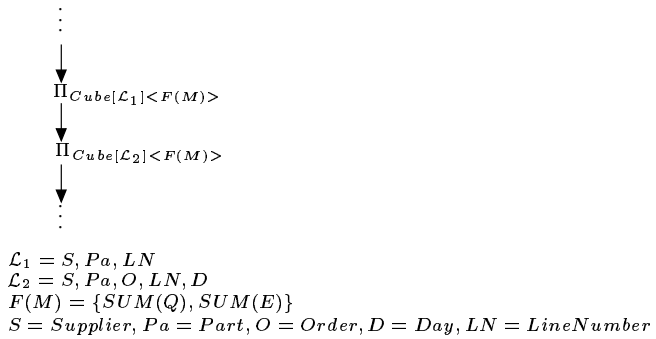


Figure 25: A Fragment of plan

The cost formula for a SQL_{XM} plan is:

$$t_{Plan} = t_{Inlinings} + t_{OLAP,TEMP}$$

where,

1. We use *cost* to represent the evaluation time in milliseconds.

2. $t_{Inlinings}$ is the cost of inlining the decoration data, if there is any inlining operators in the plan. It includes the cost of processing OLAP queries, loading XML data from external data sources and the cost of transforming the predicates, that is: $t_{Inlinings} = t_\iota + Max(t_{\tau_1}, \ldots, t_j)$, where t_ι is the cost of inlining operator, $Max(t_{\tau_1}, \ldots, \tau_j)$ is maximal cost of ι's secondary argument operators, in another word, the maximal time of loading XML data into the temporary component. In Figure 26, $t_{Inlining} = t_{\iota_{N/NL/P<9000000}} + t_{\tau_{[N/NL/P]}}$.

3. $t_{OLAP,TEMP}$ represents the cost of loading OLAP data into the temporary component plus the cost of the operations to manipulate the temporary cube, which is composed of federation operations, non-inlined XML data and dimension values retrievals, that is, $t_{OLAP,TEMP} = OverallCost(Op_n)$, where n is the number of operators on the branch from the bottom to the top of the plan, in Figure 26, $n = 5$ and $Op_5 = \iota_{[N/NL/P<9000000']}$.

   Moreover, *OverallCost*(Op_i) is a function that returns the overall cost of a subplan rooted at Op_i, $Op_i \in \{\Pi_{Fed}, \sigma_{Fed}, \iota, \phi, \Pi_{Cube}, \Sigma_{Cube}\}$,

$$OverallCost(Op_i) = \begin{cases} (1) & if \;\; Op_i \neq \phi \\ t_{OLAP,Eval} + t_{OLAP,Trans} & if \;\; Op_i = \phi \end{cases}$$

   where,

   (a) $(1) = t_{Op_i} + Max(OverallCost(Op_{i-1}), Max(t_{op_1}, \ldots, t_{op_j}))$,
   where $\{op_1, \ldots, op_j\} = SubOps(Op_i)$. *SubOps*($Op_i$) returns the secondary arguments of Op_i.

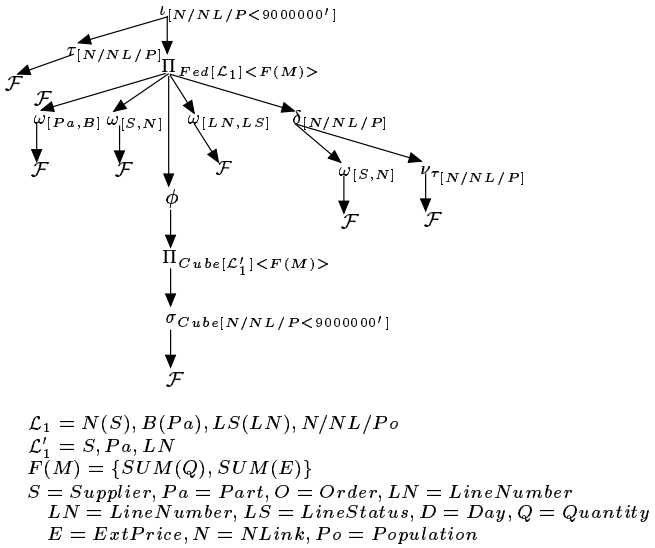


Figure 26: Example Plan for Cost Estimation

(b) $t_{OLAP,Eval}$ is the evaluation cost of OLAP queries, that is, $t_{OLAP,Eval} = t_{\Pi_{Cube}} + t_{\sigma_{Cube}}$.

(c) $t_{OLAP,Trans}$ is the cost of loading OLAP data into the temporary component. If the OLAP data and the temporary component are not physically on the same server, then it is the network transferring cost, otherwise it is the local data loading time. As for now, we only take the later one into concern, that is $t_{OLAP,Trans} = t_{\phi}$.

9.1 Statistics

The estimation of cost parameters is based on statistical information shown below, and how these values are determined are described in Section 10.2

- *OVERHEAD*, in short, *OH*, is the initial cost that is only dependent on the DBMS, such as the overhead for compiling, locking, checking for security, and etc. The overhead is repeated for each query that is posed, including the temporary component and OLAP component queries. The unit is milliseconds (ms).

- *DataReadRate*, the rate with which data can be loaded into memory from physical storage devices. It is used to estimate the cost of loading tuples into memory before they are processed. The unit is bytes/ms.

- *DataWriteRate*, the rate with which data can be written back into physical storage devices. It is used to estimate the cost of populating the processed tuples. The unit is bytes/ms.

- *InlineRate*, the rate with which the query engine generates a predicate string given some amount of inlining data.
- $Cardinality(T)$, the function which returns the cardinality of a table T. The unit is bytes/ms.
- $Selectivity(\theta, T)$, the function which returns the fraction of the table T that satisfies a given predicate θ.
- $Size(T)$, the function which returns the total size of table T. Similarly, $Size(C)$ returns the size of C, where C may be a cube resulting from an OLAP query. If C is the base cube, $Size(C)$ is the size of the fact table. The unit is bytes.
- $AttrSize(attrs)$, the function which returns the size of a tuple with the attributes in the given set $attrs$. The unit is bytes.
- $NumDistinct(attrs)$, the function which returns the number of distinct values for attributes in $attrs$.
- $FactTransRate$, the rate with which the OLAP facts are transfered from the cube to the temporary database.
- $RecordLoadRate$, the rate with which records in the temporary database can be loaded into the memory via database connections before they are processed by the query engine. It is different from $DataReadRate$ as $DataReadRate$ is the data loading speed for DBMS, while $RecordLoadRate$ is for the query engine reading data through database connections. The unit is bytes/ms.
- $FactSize(M)$, this function returns the size in bytes of a fact containing only values for the measures in M.
- $RollUpFraction(\mathcal{L}, C)$, a function returns the fraction to which cube C is reduced in size, when it is rolled up to the levels $\mathcal{L}$.

9.2 Cost Formulas for SQL Operators

A federation operator is probably composed of several regular SQL operations, e.g. a federation selection may first join several tables and then performs a regular selection over the join result. We first give the cost formulas for the regular SQL operations that are performed in the temporary component, then the compositions of SQL operation that are the interpretations of federation operators, and finally the cost formulas for the federation operators. In the following context, let T, T_1, T_2 be the tables participating the operations, θ be a selection or join predicate, $attrs$ be the target attributes in SELECT clause.

9.2.1 SQL Selection

According to the cost formulas defined in [8], we divide the select cost in the following three components: initialization cost, cost to find qualifying tuples and cost to produce selected tuples. We assume the CPU time for checking the selection condition for a tuple can be ignored in comparison with the IO cost of fetching a tuple, thus the second component is the cost of loading all the candidate tuples.

Through the observations on SQL Server, it is found that the algorithm used in retrieving rows from the temporary tables is always "Table Scan", therefore we use the formula below to estimate the cost of a SQL selection.

$$t_{\sigma_\theta(T)} = OH + \frac{Size(T)}{DataReadRate} + \frac{Selectivity(\theta, T) \times Size(T)}{DataWriteRate}$$

If a table is revisited by a selection, then the initialization cost can be subtracted from the above equation, that is: $t'_{\sigma_\theta(T)} = t_{\sigma_\theta(T)} - OH$. Moreover, if the cardinality of the table is small enough, the cost of reading the tuples can be ignored, too.

The cardinality of the resulting table, T', is:

$$c_{\sigma_\theta(T)} = Cardinality(T) \times Selectivity(\theta, T)$$

Example 9.1 A SQL query below is used to select all the tuples in the table shown in Figure 7.

```
SELECT   *
INTO     Temptable
FROM     T1
```

Let *OH*=130 ms,*DataReadRate*=11065 bytes/ms, *DataWriteRate*=9000 bytes/ms, *Size*(T1)=200 bytes, *Cardinality*(T1)=4 and *Selectivity*=100% (See Section 10.2 for statistics retrieval). The cost for this query is:

$$t_{\sigma_{true}(T1)} = 130\,ms + \frac{200\,bytes}{11065\,bytes/ms} + \frac{1 \times 200\,bytes}{9000\,bytes/ms} \approx 130.040\,ms$$

The cardinality is: $c_{\sigma_{true}(T1)} = 4 \times 100\% = 4$.

9.2.2 SQL Projection

The SQL projection selects all the rows but with only certain attributes. The cost formula is similar to that of SQL selection. That is,

$$t_{\pi_{attrs}(T)} = OH + \frac{Size(T)}{DataReadRate} + \frac{Cardinality(T) \times AttrSize(attrs)}{DataWriteRate}$$

The cardinality of a table after projection is not changed, thus the cardinality of the resulting table is:

$$c_{\pi_{attrs}}(T) = Cardinality(T)$$

Example 9.2 A SQL query below is used to select the attribute `Nation` from all the tuples in the table shown in Figure 7.

```
SELECT   Nation
INTO     Temptable
FROM     T1
```

Let *OH*=130 ms,*DataReadRate*=11065 bytes/ms, *DataWriteRate*=9000 bytes/ms, *Size*(T1)=400 bytes, *Cardinality*=4, and *AttrSize*({Nation})=25 bytes. The cost for this query is:

$$t_{\pi_{\{Nation\}}(T1)} = 130\,ms + \frac{400\,bytes}{11065\,bytes/ms} + \frac{4 \times 25\,bytes}{9000\,bytes/ms} \approx 130.047\,ms$$

The cardinality is: $c_{\pi_{\{Nation\}}(T1)} = 4$.

9.2.3 SQL Aggregation

The SQL aggregation first groups the tuples and then applies aggregation functions to the groups. As described by [6], for aggregation operators, `the hash match uses the input to build the hash table (removing duplicates and computing any aggregate expressions). When the hash table is built, scan the table and output all entries`. Based on this, an aggregation is decomposed into several stages. An aggregation first scans the tuples and then the tuples containing the attributes on which to group and the to-be-aggregated attributes are partitioned using a hash function and written back to disk, during this, aggregation functions are performed on the fly. After the table is built, each partition is then read; values of the aggregated attributes and the attributes identifying the groups are written into a result table.

The cost formula is shown below:

$$t_{_{attrs_1}\mathcal{G}_{F<attrs_2>}(T)} = OH + \frac{Size(T)}{DataReadRate} + \frac{Cardinality(T) \times AttrSize(attrs_1 \cup attrs_2)}{DataWriteRate} + \frac{Cardinality(T) \times AttrSize(attrs_1 \cup attrs_2)}{DataReadRate} + \frac{NumDistinct(attrs_1) \times AttrSize(attrs_1 \cup attrs_2)}{DataWriteRate}$$

where $attrs$ represents the attributes on which to group, $F < attrs_2 >$ represents the aggregate functions on the attributes to be calculated.

The cardinality of the resulting table is:

$$c_{_{attrs_1}\mathcal{G}_{F<attrs_2>}(T)} = NumDistinct(attrs_1)$$

Example 9.3 A SQL query below is used to aggregate table T1 on attributes `Supplier`. T1 is shown in Figure 6.

```
SELECT     SUM(Quantity),SUM(ExtPrice),Supplier
INTO       Temptable
FROM       T1
GROUP BY   Nation
```

Let *OH*=130 ms,*DataReadRate*=11065 bytes/ms, *DataWriteRate*=9000 bytes/ms, *Size*(T1)=415 bytes, *Cardinality*(T1)=5, *AttrSize*({Supplier})=25 bytes, *AttrSize*({Quantity,ExtPrice})=18 bytes, and *NumDistinct*({Supplier})=3. The cost for this query is:

$$\begin{aligned} t_{\{Supplier\}\mathcal{G}_{SUM(Quantity),SUM(ExtPrice)}(T1)} &= 130\,ms + \frac{415\,bytes}{11065\,bytes/ms} + \frac{5 \times 43\,bytes}{9000\,bytes/ms} \\ &\quad + \frac{5 \times 43\,bytes}{11065\,bytes/ms} + \frac{3 \times 43\,bytes}{9000\,bytes/ms} \\ &\approx 130.095\,ms \end{aligned}$$

The cardinality is: $c_{\pi_{\{Nation\}}(T1)} = 3$.

9.2.4 SQL Join

Through a large quantity of observations, we have found the physical algorithm for a join operation in the temporary component is a hash join with sequential scans on both of the participating tables. As described by [6], the hash join algorithm uses the first input (smaller) table to build the hash table and the second input to probe the hash table, and then outputs the matches. Based on this, a hash join is decomposed into several stages in the cost formula. A hash join first builds a hash table for the smaller table in size, which includes a table scan and building a hash table on disk. Then, it uses the other table to probe the hash table, and at last outputs the matches.

The cost formula for a hash join of two table is:

$$t_{T_1 \bowtie_{\theta_1} T_2} = OH + \frac{Size(T_1)}{DataReadRate} + \frac{Size(T_2)}{DataReadRate} + \frac{Size(T_1)}{DataWriteRate} + \frac{Size(T_1)}{DataReadRate} + \frac{Cardinality(T_1) \times Cardinality(T_2) \times Selectivity(\theta_1, T_1 \times T_2) \times AttrSize(attrs_{T_1 \bowtie_{\theta_1} T_2})}{DataWriteRate}$$

where, T_1 is the smaller table in size, θ_1 is the join predicate, $Selectivity(\theta_1, T_1 \times T_2)$ returns the join selectivity of θ_1 for table T_1 and T_2. A Join without specified join predicates is a natural join, the join key is the common attributes.

The cardinality of the resulting table is:

$$c_{T_1 \bowtie_{\theta_1} T_2} = Cardinality(T_1) \times Cardinality(T_2) \times Selectivity(\theta_1, T_1 \times T_2)$$

For a join of n ($n \geq 3$) tables, two tables are first joined, which can be estimated as $t_{T_1 \bowtie_{\theta_1} T_2}$. Then the third table is joined with the result table, the new result is joined again with the fourth table. The join process repeats until it reaches the n-th table. In the optimizer of a DBMS, producing the join path is a complex process. Therefore, in our current optimizer, only a rough cost estimation is provided.

The cost formula is:

$$t_{T_n \bowtie_{\theta_{n-1}} TT_{n-1}} = t_{TT_{n-1}} + \frac{Size(TT_{n-1})}{DataReadRate} + \frac{Cardinality(T_n) \times Cardinality(TT_{n-1}) \times Selectivity(\theta_{n-1}, T_n \times TT_{n-1})}{DataWriteRate} \times AttrSize(attrs_{T_n \bowtie_{\theta_{n-1}} TT_{n-1}})$$

where,

$$TT_{n-1} = \begin{cases} T_{n-1} \bowtie_{\theta_{n-1}} TT_{n-2} & \text{if } n > 3 \\ T_2 \bowtie_{\theta_1} T_1 & \text{if } n = 3 \end{cases}$$

, θ_{n-1} is the join predicate for T_n and TT_{n-1}, and $Selectivity(\theta_{n-1}, T_n \times TT_{n-1})$ returns the join selectivity.

The cardinality of the resulting table is:

$$c_{T_n \bowtie_{\theta_{n-1}} TT_{n-1}} = Cardinality(T_n) \times Cardinality(TT_{n-1}) \times Selectivity(\theta_{n-1}, T_n \times TT_{n-1})$$

Example 9.4 A SQL query below is used to join table T1 in Figure 7 and table T2 in Figure 8.

```
SELECT   T1.Supplier s, T1.Nation n1, T2.Nation n2, T2.Population p
INTO     Temptable
FROM     T1 INNER JOIN T2
         ON T1.Nation=T2.Nation
```

Let *OH*=130 ms,*DataReadRate*=11065 bytes/ms, *DataWriteRate*=9000 bytes/ms, *Size*(T1)=200 bytes, *Size*(T2)=102 bytes, *Cardinality*(T1)=4, and *Cardinality*(T2)=3 . The resulting table has four attributes, therefore the size of each row is: *AttrSize*({s, n1, n2, p})=84 bytes. The join yields a table of four tuples, therefore, *Selectivity*(T1.Nation=T2.Nation,$T1 \times T2$)=33.3%. The cost for this query is:

$$\begin{aligned} t_{T1 \bowtie_{T1.Nation=T2.Nation} T2} &= 130\,ms + \frac{200\,bytes}{11065\,ms} + \frac{102\,bytes}{11065\,bytes/ms} + \frac{102\,bytes}{9000\,bytes/ms} \\ &\quad + \frac{102\,bytes}{11065\,bytes/ms} + \frac{4 \times 3 \times 33.3\% \times 59\,bytes}{9000\,bytes/ms} \\ &\approx 130.074\,ms \end{aligned}$$

The cardinality of the resulting table is: $c_{T1 \bowtie_{T1.Nation=T2.Nation} T2} = 4 \times 3 \times 33.3\% \approx 4$

9.2.5 Composition of SQL Join and Selection

A composition of join and selection will be used by federation selection operators. This composite operation means only the tuples that satisfy certain conditions may participate a join. Thus the cost formula of a join with select conditions on both tables is:

$$t_{\sigma_{\theta_{T_1}}(T_1) \bowtie_{\theta_1} \sigma_{\theta_{T_2}}(T_2)} = OH + \frac{Size(T_1)}{DataReadRate} + \frac{Size(T_2)}{DiskReadReate} + \frac{Selectivity(\theta_{T_1}, T_1) \times Size(T_1)}{DataWriteRate} + \frac{Selectivity(\theta_{T_1}, T_1) \times Size(T_1)}{DataReadRate} + \frac{Cardinality(T_1) \times Cardinality(T_2) \times S_{[\theta_1, \theta_{T_1}, \theta_{T_2}]} \times AttrSize(attrs_{T_1 \bowtie_{\theta_1} T_2})}{DataWriteRate}$$

where, $S_{[\theta_1,\theta_{T_1},\theta_{T_2}]} = Selectivity(\theta_{T_1}, T_1) \times Selectivity(\theta_{T_2}, T_2) \times Selectivity(\theta_1, T_1 \times T_2)$, θ_{T_1} and θ_{T_2} are the select predicates on T_1 and T_2 respectively, and θ_1 is the join predicate.

For a join of n ($n \geq 3$) tables, the cost is:

$$t_{\sigma_{\theta_{T_n}}(T_n)\bowtie_{\theta_{n-1}} TT_{n-1}} = t_{TT_{n-1}} + \frac{Selectivity(\theta_{TT_{n-1}}, TT_{n-1}) \times Size(TT_{n-1})}{DataReadRate} + \frac{Cardinality(T_n) \times Cardinality(TT_{n-1}) \times S_{[\theta_{n-1},\theta_{T_n}]}}{DataWriteRate} \times AttrSize(attrs_{T_n \bowtie_{\theta_{n-1}} TT_{n-1}})$$

where,

$$TT_{n-1} = \begin{cases} \sigma_{\theta_{T_{n-1}}}(T_{n-1}) \bowtie_{\theta_{n-2}} TT_{n-2} & \text{if } n > 3 \\ \sigma_{\theta_{T_1}}(T_1) \bowtie_{\theta_1} \sigma_{\theta_{T_2}}(T_2) & \text{if } n = 3 \end{cases}$$

,$S_{[\theta_{n-1},\theta_{T_n}]} = Selectivity(\theta_{T_n}, T_n) \times Selectivity(\theta_{n-1}, T_n \times TT_{n-1})$, θ_{T_n} and $\theta_{TT_{n-1}}$ are the select predicates on T_n and TT_{n-1} respectively, and θ_{n-1} is the join predicate.

The cardinality of the resulting table is:

$$c_{\sigma_{\theta_{T_n}}(T_n)\bowtie_{\theta_{n-1}} TT_{n-1}} = Cardinality(T_n) \times Cardinality(T_2) \times S_{[\theta_{n-1},\theta_{T_n}]}$$

Example 9.5 A SQL query below is used to join the satisfying tuples from table T1 in Figure 7 and table T2 in Figure 8.

```
SELECT   T1.Supplier s, T1.Nation n1, T2.Nation n2, T2.Population
INTO     Temptable
FROM     T1 INNER JOIN T2
         ON T1.Nation=T2.Nation
WHERE    T2.Population<20
```

As the query has no constraints on tuples in T1, the selectivity is 100%. In addition to the same statistics used in the last example, the required cost parameters includes the selectivity of the predicate on T2, which is *Selectivity*(T2.Population, T2)=66.7%. The cost for this query is:

$$\begin{aligned} t_{T1\bowtie_{T1.Nation=T2.Nation}\sigma_{T2.Population<20}(T2)} &= 130\,ms + \frac{200\,bytes}{11065\,ms} + \frac{102\,bytes}{11065\,bytes/ms} \\ &\quad + \frac{102\,bytes \times 66.7\%}{9000\,bytes/ms} + \frac{102\,bytes \times 66.7\%}{11065\,bytes/ms} \\ &\quad + \frac{4 \times 3 \times 33.3\% \times 66.7\% \times 59\,bytes}{9000\,bytes/ms} \\ &\approx 130.058\,ms \end{aligned}$$

The cardinality of the resulting table is: $c_{T1\bowtie_{T1.Nation=T2.Nation}T2} = 4 \times 3 \times 33.3\% \approx 4$

9.2.6 Composition of SQL Projection, Join and Selection

A composition of SQL projection, join and selection is sometimes a direct translation of a federation selection operator. This composite operation first selects the tuples, then join them and finally projects away unnecessary attributes.

$$t_{\pi_{attrs}(\sigma_{\theta_{T_1}}(T_1)\bowtie_{\theta_1}\sigma_{\theta_{T_2}}(T_2))} = OH + \frac{Size(T_1)}{DataReadRate} + \frac{Size(T_2)}{DataReadRate} + \frac{Selectivity(\theta_{T_1}, T_1) \times Size(T_1)}{DataWriteRate} + \frac{Selectivity(\theta_{T_1}, T_1) \times Size(T_1)}{DataReadRate} + \frac{Cardinality(T_1) \times Cardinality(T_2) \times S_{[\theta_1,\theta_{T_1},\theta_{T_2}]} \times AttrSize(attrs)}{DataWriteRate}$$

where, $S_{[\theta_1,\theta_{T_1},\theta_{T_2}]} = Selectivity(\theta_{T_1}, T_1) \times Selectivity(\theta_{T_2}, T_2) \times Selectivity(\theta_1, T_1 \times T_2)$, θ_{T_1} and θ_{T_2} are the select predicates on T_1 and T_2 respectively, and θ_1 is the join predicate.

For a projection on a join of n ($n \geq 3$) tables, the cost is:

$$t_{\pi_{attrs}(\sigma_{\theta_{T_n}}(T_n)\bowtie_{\theta_{n-1}}TT_{n-1})} = t_{TT_{n-1}} + \frac{Size(TT_{n-1})}{DataReadRate} + \frac{Cardinality(T_n) \times Cardinality(TT_{n-1}) \times S_{[\theta_{n-1},\theta_{T_n}]} \times AttrSize(attrs)}{DataWriteRate}$$

where,

$$TT_{n-1} = \begin{cases} \sigma_{\theta_{T_{n-1}}}(T_{n-1}) \bowtie_{\theta_{n-2}} TT_{n-2} & \text{if } n > 3 \\ \sigma_{\theta_{T_1}}(T_1) \bowtie_{\theta_1} \sigma_{\theta_{T_2}}(T_2)) & \text{if } n = 3 \end{cases}$$

,$S_{[\theta_{n-1},\theta_{T_n}]} = Selectivity(\theta_{T_n}, T_n) \times Selectivity(\theta_{n-1}, T_n \times TT_{n-1})$, θ_{T_n} and $\theta_{TT_{n-1}}$ are the select predicates on T_n and TT_{n-1} respectively, and θ_{n-1} is the join predicate.

The cardinality of the resulting table is:

$$c_{\pi_{attrs}(\sigma_{\theta_{T_n}}(T_n)\bowtie_{\theta_{n-1}}TT_{n-1})} = Cardinality(T_n) \times Cardinality(TT_{n-1}) \times S_{[\theta_{n-1},\theta_{T_n}]}$$

Example 9.6 A SQL query below is used to select values of interesting attributes from the join of the satisfying tuples from table T1 in Figure 7 and table T2 in Figure 6.

```
SELECT   T2.*
INTO     Temptable
FROM     T1 INNER JOIN T2
         ON T1.Supplier=T2.Supplier
WHERE    T1.Nation='DK'
```

Let *OH*=130 ms,*DataReadRate*=11065 bytes/ms, *DataWriteRate*=9000 bytes/ms, *Size*(T1)=200 bytes, *Size*(T2)=415 bytes, *Cardinality*(T1)=4, *Cardinality*(T2)=5, *AttrSize*(Attrs(T2))=415 bytes and *Selectivity*(T1.Nation='DK',T1)=50%. The join selectivity *Selectivity*(T1.Supplier=T2.Supplier, T1$\times$ T2)=25%. Since the query has no constraints on the tuples in T2, the selectivity on T2 is 100%.

Then the cost for this query is:

$$\begin{aligned} t_{T2\bowtie_{T1.Supplier=T2.Supplier}\sigma_{T1.Nation='DK'}(T1)} &= 130\,ms + \frac{200\,bytes}{11065\,ms} + \frac{415\,bytes}{11065\,bytes/ms} \\ &\quad + \frac{200\,bytes \times 50\%}{9000\,bytes/ms} + \frac{200\,bytes \times 50\%}{11065\,bytes/ms} \\ &\quad + \frac{4 \times 5 \times 25\% \times 50\% \times 415\,bytes}{9000\,bytes/ms} \\ &\approx 130.191\,ms \end{aligned}$$

The cardinality of the resulting table is: $c_{T2\bowtie_{T1.Supplier=T2.Supplier}\sigma_{T1.Nation='DK'}(T2)} = 4 \times 5 \times 25\% \times 50 \approx 2$

9.2.7 Composition of SQL Aggregation and Join

A composition of SQL aggregation and join may be used by a federation generalized projection. This composite operation first joins tables and then aggregate the resulting table.

$$t_{attrs_1\mathcal{G}_{F<attrs_2>}(T_1\bowtie_{\theta_1}T_2)} = t_{T_1\bowtie_{\theta_1}T_2} + \frac{Size(T')}{DataReadRate} + \frac{NumDistinct(attrs_1) \times AttrSize(attrs_1 \cup attrs_2)}{DataWriteRate}$$

where, $T' = T_1 \bowtie_{\theta_1} T_2$, $attrs_1$ represents the attributes on which to group, and $F < attrs_2 >$ represents the aggregate functions on the attributes to be calculated.

For the join where the number of participating tables is n ($n \geq 3$), the cost is:

$$t_{attrs_1\mathcal{G}_{F<attrs_2>}(T_n\bowtie_{\theta_{n-1}}TT_{n-1})} = t_{T_n\bowtie_{\theta_{n-1}}TT_{n-1}} + \frac{Size(T')}{DataReadRate} + \frac{NumDistinct(attrs_1) \times AttrSize(attrs_1 \cup attrs_2)}{DataWriteRate}$$

where, $T' = T_1 \bowtie_{\theta_1} T_2 \bowtie_{\theta_2} \ldots \bowtie_{\theta_{n-1}} T_n$,

The cardinality of the resulting table is:

$$c_{attrs_1\mathcal{G}_{F<attrs_2>}(T_1\bowtie_{\theta_1}T_2\bowtie_{\theta_2}\ldots\bowtie_{\theta_{n-1}}T_n)} = NumDistinct(attrs_1)$$

Example 9.7 A SQL query below is the composition of these two SQL operations. Table T1 is shown in Figure 7. The other participating table T2 is shown in Figure 6.

```
SELECT      SUM(Quantity), SUM(ExtPrice), T1.Nation
INTO        Temptable
FROM        T1 INNER JOIN T2
            ON T1.Supplier=T2.Supplier
GROUP BY    T1.Nation
```

Let *OH*=130 ms,*DataReadRate*=11065 bytes/ms, *DataWriteRate*=9000 bytes/ms, *Size*(T1)=200 bytes, *Size*(T2)=415 bytes, *Cardinality*(T1)=4, *Cardinality*(T2)=5, and the join selectivity *Selectivity*(T1.Supplier=T2.Supplier, T1$\times$ T2)=25%. The intermediate table after the join has all the attributes, therefore *AttrSize*($T1 \bowtie_{T1.Supplier=T2.Supplier} T_2$)=117 bytes and *NumDistinct*(Nation)=3. The final result table has three attributes, therefore the size of each row is: *AttrSize*({Quantity, ExtPrice,Nation})=43 bytes.

First, the cost for the join is estimated.

$$\begin{aligned} t_{T1\bowtie_{T1.Supplier=T2.Supplier}T2} &= 130\,ms + \frac{415\,bytes}{11065\,ms} + \frac{200\,bytes}{11065\,bytes/ms} + \frac{200\,bytes}{9000\,bytes/ms} \\ &\quad + \frac{200\,bytes}{11065\,bytes/ms} + \frac{5 \times 4 \times 25\% \times 117\,bytes}{9000\,bytes/ms} \\ &\approx 130.161\,ms \end{aligned}$$

The full cost is:

$$\begin{aligned} t_{Nation\mathcal{G}_{SUM(Quantity),SUM(ExtPrice)}T1\bowtie_{T1.Supplier=T2.Supplier}T2} &= 130.161\,ms + \frac{585\,bytes}{11065\,ms} \\ &\quad + \frac{3 \times 43\,bytes}{9000\,bytes/ms} \\ &\approx 130.228\,ms \end{aligned}$$

The cardinality of the resulting table is: $c_{T1\bowtie_{T1.Supplier=T2.Supplier}T2} = NumDistinct(Nation) = 3$.

9.3 Cost Formulas for Federation operators

9.3.1 Inlining

As defined in Section 9.3.1, an inlining operator may have several predicates in its parameter set. Thus The cost of an inlining operator is the maximal cost of inlining level expressions for the predicates. Each inlining comprises the cost of loading data into memory, and the CPU time to generate the predicate string. That is:

$$t_{l_{[\theta_1,\ldots,\theta_n]}} = Max(t_{Inlining,\theta_1}, \ldots, t_{Inlining,\theta_n})$$

, where $t_{Inlining,\theta_i} = t_{LoadIntoMem} + t_{GenString}$.

Following the definition of transforming predicates, in Section 9.3.1, the cost formulas can be grouped as below where θ is a predicate, the binary operator *bo* is either AND or OR, and the predicate operator *po* is one of : $=, <, >, <>, >=, <=$ and LIKE.

1. if $\theta = l/Link/xp\ po\ K$, where K is a constant value, the cost is:

$$t_{Inlining,\theta} = OH + \frac{Size(T)}{RecordLoadRate} + \frac{Selectivity(\theta', T) \times Cardinality(T) \times AttrSize(\{l\})}{InlineRate}$$

where T is the table generated by $\tau_{[l/Link/xp]}$, and $\theta' = xp\ po\ K$.

2. if $\theta = l_1/Link/xp\ po\ l_2$, where l_2 is a level, the cost is:

$$t_{Inlining,\theta} = OH + \frac{Size(T)}{RecordLoadRate} + \frac{Cardinality(T) \times AttrSize(\{l, xp\})}{InlineRate} \quad (32)$$

where T is the table generated by $\tau_{[l_1/Link/xp]}$. Notice that the selection predicate now is $true$ because we need all the tuples in the temporary table.

3. if $\theta = l/Link/xp\ po\ M$, where M is a measure. The cost formula is similar to Equation 32.

4. if $\theta = l_1/Link_1/xp_1\ po\ l_2/Link_2/xp_2$, the cost is:

$$t_{Inlining,\theta} = 2 \times OH + \frac{Size(T_1)}{RecordLoadRate} + \frac{Size(T_2)}{RecordLoadRate} + \frac{Cardinality(T_1) \times Cardinality(T_2) \times AttrSize(\{l_1, l_2, xp_1, xp_2\})}{InlineRate}$$

where T_1, T_2 are the tables generated by $\tau_{[l_1/Link_1/xp_1]}$, $\tau_{[l_2/Link_2/xp_2]}$, respectively . Notice that the selection predicate now is $true$ because we need all the tuples in the temporary table.

5. if $\theta = l/Link/xp\ IN\ (K_1, \ldots, K_n)$, where K_i is a constant value, the cost is:

$$t_{Inlining,\theta} = t_{Inlining,l/Link/xp=K_1} + \ldots + t_{Inlining,l/Link/xp=K_n}$$

6. if $\theta = NOT\ \theta_1$, the cost is:

$$t_{Inlining,\theta} = t_{Inlining,\theta_1}$$

7. if $\theta = \theta_1\ bo\ \theta_2$, the cost is:

$$t_{Inlining,\theta} = t_{Inlining,\theta_1} + t_{Inlining,\theta_2}$$

Example 9.8 Suppose we have a natural link,TLink = (”Type”, ”types.xml”, ”/Types/Type/”, ”TypeName”), $\tau_{[Type/TLink/RetailPrice]}(\mathcal{F})$ yields a temporary table in Figure 27(a) being added to the temporary cube, and $\iota_{[Type/TLink/RetailPrice<1600]}(\mathcal{F})$ generates the new predicate string: “Type in (’Copper’,’Tin’,’Steel’,’Nickel’)”. Let *OH*=130 ms,*RecordLoadRate*=400 bytes/ms, *InlineRate*=2350 bytes/ms, *Size*(Typeprices)=170 bytes, *AttrSize*({Type})= 25bytes, *Selectivity*=80% and *Cardinality*(Typeprices)=5. According to the first transformation cost, the inlining cost is:

$$t_{Inlining,Type/TLink/RetailPrice<1600} = 130\,ms + \frac{170\,bytes}{400\,bytes/ms} + \frac{0.8 \times 5 \times 25\,bytes}{2350\,bytes} \approx 130.468\,ms$$

TypeName	RetailPrice
Brass	1800
Copper	1038
Tin	1753
Steel	1594
Nickel	1491

(a) Typeprices

Nation	Population
N1	100
N2	200
⋮	⋮
N8	800
N9	900

(b) Populations

Figure 27: Temporary Decoration Tables

Suppose another XML transfer $\tau_{[Nation/NLink/Population]}(\mathcal{F})$ yields a table in Figure 27(b). $\iota_{[Type/TLink/RetailPrice<1600\ AND\ Nation/NLink/Population<500]}(\mathcal{F})$ generates a predicate string: "Type in ('Copper','Tin','Steel','Nickel') AND Nation in ('N1','N2','N3','N4','N5','N6')". Let *Size*(Populations)=306 bytes, *AttrSize*($\{$Nation$\}$)= 25bytes, *Cardinality*(Population)=9 and *Selectivity*=56%. The cost of writing the full string is

$$t_{Inlining,Type/TLink/RetailPrice<1600\ AND\ Nation/NLink/Population<500} = t_{Inlining,Type/TLink/RetailPrice<1600} + t_{Inlining,Nation/NLink/Population<500}$$

where,$t_{Inlining,Type/TLink/RetailPrice<1600} \approx 130.468\,ms$ and $t_{Inlining,Nation/NLink/Population<500} = 130\,ms + \frac{306\,bytes}{400\,bytes/ms} + \frac{0.56 \times 9 \times 25\,bytes}{2350\,bytes}$ $\approx 130.819\,ms$. Thus the total cost is $261.287\,ms$.

9.3.2 Decoration

A decoration create a dimension table which is a join between two tables created by a dimension transfer and an XML transfer, thus it is actually a regular SQL join. The cost of a decoration is:

$$t_{\delta_{l_z/Link/xp}} = t_{T_1 \bowtie_\theta T_2}$$

where T_1 is the temporary table created by $\omega_{[\perp_z,l_z]}$, T_2 is the table created by $\tau_{[l_z/Link/xp]}$, and θ is the join predicate.

The cardinality of the resulting table is:

$$c_{\delta_{l_z/Link/xp}} = c_{T_1 \bowtie_\theta T_2}$$

Example 9.9 See Example 9.4, where T1 is the temporary table created by $\omega_{[Nation,Supplier]}$, T2 is the table created by $\tau_{[Nation/NLink/Population]}$, and $\theta =" T1.Nation = T2.Nation"$.

9.3.3 Federation Selection

A federation selection is a regular SQL selection if the fact table contains all the levels and measures referenced in the predicates, otherwise tuples containing satisfying values of the required levels is needed to be joined with the fact table. Note that the join keys are all bottom levels which must exist in the fact table, therefore natural joins are performed. The cost formula is:

$$t_{\sigma_{Fed[\theta]}(\mathcal{F})} = \begin{cases} t_{\sigma_\theta(F)} & ReferredAttr(\theta) \subseteq Attrs(F). \\ t_{\pi_{Attrs(T)}(\sigma_{\theta_1}(T_1) \bowtie \ldots \bowtie \sigma_{\theta_n}(T_n) \bowtie \sigma_{\theta_{\mathcal{F}}}(F))} & if\ \exists l \in ReferredAttr(\theta_i)(l \in Attrs(T_i) \wedge \\ & \qquad l \notin Attrs(F)). \end{cases}$$

where F is the fact table, $ReferredAttr(\theta)$ is a function that returns the attributes referred by θ, $Attrs(T_i)$ is a function that returns the attributes in T_i, $*$ represents all the attributes of the table. In the second case, $\theta = \theta_1 \wedge \ldots \wedge \theta_n \wedge \theta_{\mathcal{F}}$.

The cardinality of the resulting table can be computed through the base SQL operations.

Example 9.10 For $\sigma_{Fed[Nation='DK']}(\mathcal{F})$, *Nation* is a non-bottom level not present in the fact table. Thus the temporary table containing values of *Nation* is needed to be joined with the fact table F. In Example 9.6, T1 is the temporary table created by $\omega_{[Nation,Supplier]}$, T2 is the Fact table, the cost and cardinality are estimated by the cost formula for composition of projection, join and selection.

9.3.4 Federation Generalized Projection

A federation GP is a regular SQL projection if the fact table contains all the levels and measures referenced in the SELECT clause, otherwise a table containing the required levels is first needed to be joined with the fact table.

The cost formula is:

$$t_{\pi_{Fed[\mathcal{L}]<F(M)>}(\mathcal{F})} = \begin{cases} (1) & \text{if } \mathcal{L} \subseteq Attrs(F) \wedge M \neq \phi. \\ (2) & \text{if } \exists l \in \mathcal{L}(l \in Attrs(T_i) \wedge l \notin (Attrs(F)) \wedge M \neq \phi. \end{cases}$$

where, $(1) = t_{\mathcal{L}\mathcal{G}_{F<M>}(F)}$, and $(2) = t_{\mathcal{L}\mathcal{G}_{F<M>}(T_1 \bowtie \ldots \bowtie T_n \bowtie F)}$.

The cardinality of the resulting table can be computed through the base SQL operations.

Example 9.11 For $\Pi_{Fed[\{Nation\}]<SUM(Quantity),SUM(ExtPrice)>}(\mathcal{F})$, "Nation" is a non-bottom level not present in the fact table. Thus, the temporary table containing values of "Nation" is needed to be joined with the fact table F. In Example 9.7, T2 is the fact table, T1 has the level `Nation`, these two tables are first joined then aggregated. The cost and cardinality are approximated.

9.3.5 Dimension Transfer

A dimension transfer operator transfers the dimension values to the temporary area on the same server. The dimension values are selected by sending a simple OLAP query (See Section 10.4),

therefore the overhead is considered. The cost is:

$$t_{\omega_{[attrs]}} = OH + \frac{NumDistinct(attrs) \times AttrSize(attrs)}{FactTransRate}$$

The cardinality of the dimension table is:

$$c_{\omega_{[attrs]}} = NumDistinct(attrs)$$

Example 9.12 T1 in Figure 7 is loaded by the dimension transfer, $\omega_{[Nation,Supplier]}$. Let *FactTransRate*=880 bytes/ms, *OH*=130 ms, *NumDistinct*({Nation, Supplier})=4, and *AttrSize*({Nation, Supplier})=50 bytes. The cost is:

$$t_{\omega_{[Nation,Supplier]}(\mathcal{F})} = 130\,ms + \frac{4 \times 50\,bytes}{880\,bytes/ms} \approx 130.227\,ms$$

The cardinality is: $c_{\omega_{[Nation,Supplier]}} = 4$.

9.3.6 Fact Transfer

A fact transfer operator transfers the facts to the temporary area on the same server. The cost is:

$$t_{OLAP,Trans} = \frac{Size(F)}{FactTransRate}$$

where, $Size(F)$ is the size of the facts needed to be transfered.

The cardinality of the resulting fact table is the cardinality of the table returned by the OLAP query.

9.4 Cost Formulas for Cube Operators

The cost of an OLAP query is the estimated time it takes to evaluate a query Q in the OLAP component. There is no cost formula for each cube operator, but the whole cost for the entire query.

The size of a cube is estimated using the formulas below.

$$Size(C') = \begin{cases} Selectivity(\theta, C) \times Size(C) & \text{if } \mathcal{F}' = \sigma_{Cube[\theta]}(\mathcal{F}) \wedge C \in \mathcal{F} \wedge C' \in \mathcal{F}' \\ RollUpFraction(\mathcal{L}, C) \times \frac{AttrSize(M)}{AttrSize(\{M_1,\ldots,M_m\})} \times Size(C) & \\ \quad \text{where } \{M_1, \ldots, M_m\} \text{ is all the measures,if } \mathcal{F}' = \Pi_{Cube[\mathcal{L}]<F(M)>}(\mathcal{F}) & \\ \quad \wedge C \in \mathcal{F} \wedge C' \in \mathcal{F}' & \\ \text{Size of the fact table} & \text{if } C' \text{ is the base cube.} \end{cases}$$

According to previous work on OLAP query estimation [21], 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. First, a pre-aggregated result may be available that rolls up to the same levels as the query being evaluated. For instance, the sales by month could be precomputed and the queries on this can be answered almost instantly. Second, no pre-aggregated results are used. Instead, the query

is computed entirely from the base cube. A certain quantity of facts that satisfy the select conditions are selected and then measures are aggregated. Third, pre-aggregated results can be used, but still some facts have to be scanned and aggregated to produce the entire result.

These strategies are summarized in the following formula for the cost of an OLAP query. Assume that queries are on the form $Q(C) = \Sigma_2(\Pi_{Cube[\mathcal{L}]}(\Sigma_1(\mathcal{F})))$, where Σ_i denotes a sequence on selections, C is a component of federation $\mathcal{F}$. 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.

$$t_{OLAP,Eval} = \begin{cases} \dfrac{d}{DiskDateRate} & \text{where } d = S_{\mathcal{L}} \times Size(\Pi_{Fed[\mathcal{L}]<F(M)>}(\mathcal{F})) \\ & \text{if no selections in } \Sigma_1 \text{ refer to measures or levels not in} \\ & \Pi_{Cube[\mathcal{L}]<F(M)>}(\mathcal{F}), \text{ and } \Pi_{Cube[\mathcal{L}]<F(M)>} \text{ is pre-aggregated.} \\ \dfrac{d + 2d'}{DiskDateRate} & \text{where } d = S_{\mathcal{L}} \times Size(\Pi_{Cube[\mathcal{L}]<F(M)>}(\mathcal{F})) \text{ and} \\ & d' = S_M \times d, \text{ if any selections in } \Sigma_1 \text{ refer to measures.} \\ \dfrac{3d}{DiskDateRate} & \text{where } d = S_{\mathcal{L}} \times Size(\Pi_{Fed[\mathcal{L}']<F(M)>}(\mathcal{F})) \\ & \text{for } \Pi_{Fed[\mathcal{L}]<F(M)>}(\mathcal{F}) \rightarrow \Pi_{Fed[\mathcal{L}]<F(M)>}(\Pi_{Cube[\mathcal{L}']<F(M')>}(\mathcal{F})) \\ & \text{if no selections in } \Sigma_1 \text{ refer to measures or levels not in} \\ & \Pi_{Cube[\mathcal{L}']<F(M')>}(\mathcal{F}), \text{ and } \Pi_{Fed[\mathcal{L}']<F(M')>} \text{ is pre-aggregated.} \end{cases}$$

The cardinality of the resulting fact table is:

$$Cardinality(F') = \begin{cases} Selectivity(\theta, C) \times Cardinality(F) \\ \quad \text{if } \mathcal{F}' = \sigma_{Cube[\theta]}(\mathcal{F}) \wedge F \in C \wedge C \in \mathcal{F} \wedge F' \in C' \wedge C' \in \mathcal{F}' \\ RollUpFraction(\mathcal{L}, C) \times Cardinality(F) \\ \quad \text{if } \mathcal{F}' = \Pi_{Cube[\mathcal{L}]<F(M)>}(\mathcal{F}) \wedge F \in C \wedge C \in \mathcal{F} \wedge F \in C' \wedge C' \in \mathcal{F}' \\ Cardinality(F) \\ \quad \text{if } F' = F \end{cases}$$

9.5 XML Component Querying

As pointed out by [21], estimating cost of XML components is exceeding difficult. Hence, only a rough cost estimate can be made. The cost model is based on estimating the amount of data returned by a query, and assuming a constant data rate when retrieving data from the component. The cost model is composed of the constant overhead of performing a query and the time it takes to process the query. That is:

$$t_\tau = XMLOVERHEAD + t_{XML,Proc}$$

9.5.1 Statistics

Estimation of the cost depends on the statistical information described in the following.

1. *XMLOVERHEAD*, is the initial cost including the time spent on everything from parsing the entire XML document to parsing only the query.

2. $NodeSize(path_E)$, 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. The unit is bytes.

3. $Cardinality(path_E)$, the total number of elements pointed to by $path_E$.

4. $XMLLoadRate(x)$, the average amount of data that can be retrieved from the XML document x per millisecond.

9.5.2 Cost Formula for XML Querying

Using the statistical information above, $t_{XML,Proc}$ is calculated for an XPATH expression xp in a document x as follows:

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

The cardinality of the temporary table for decoration values is shown below.

$$c_\tau = Cardinality(xp)$$

10 Implementation

This section describes the implementation of the optimized query engine, including an example implementation of a transformation rule, getting the statistical information, the cost estimation of a plan and evaluation. Transformation rules reside in the *query rewriter*, called by function *matchAndApplyRules* (Section 6.1) in the *query rewriter* to rewrite a query plan. Statistical information is used by the cost estimation functions during the plan space enumeration to give estimated evaluation time and size. Cost evaluation is another module that gathers real data from underlying data sources and performs federation operations on the temporary data to give the final result.

The prototype is programmed in Microsoft Visual J++ 6.0 environment. Visual J++ gives the built-in Java APIs to quickly build Java applications and has a good support for *Microsoft ActiveX controls*. Using ActiveX, Java developers can write applications in Java that expose *Component Object Model* (COM) interfaces and that use COM interfaces, therefore Java applications can easily take the advantages of existing business programs working on windows platforms.

10.1 A Transformation Rule

A transformation rule in the query engine is a piece of code that consists of two components. The first determines the applicability of the rule, while the second applies the rule and produces a new

plan. The pseudo-code for Rule 7.7 is outlined briefly in Figure 28. $root$ is the reference to the root operator of a plan tree. Line 2 and 3 determine whether the rule is applicable to the plan rooted at $root$, i.e. whether the plan matches the rule. If not, the rule returns null. If it does, Line 6,7,8,9 create the new operators that are needed by the rule body (the other side of the rule that reconstructs the plan). Line 10, 11, 12 positions these operators by the alignment defined in the rule body. Finally, Line 13 returns the new plan's root.

```
Operator rule(Operator root)
1)  {
2)      if root is a federation selection
3)      {
4)          if root's child operator is fact-transfer
5)          {
6)              create a null operator nullOp;
7)              copy the federation selection's secondary child operators and
                assign the references to nullOp;
8)              create a cube selection operator cubeOp;
9)              create a copy of the subplan below the federation selection;
                /* factTrans is the fact-transfer in the copy of the subplan */
                /* restOfPlan is the sub-plan below factTrans */
10)             make factTrans the child operator of nullOp by reference;
11)             for factTrans, replace the reference to restOfPlan with that of cubeOp ;
12)             make restOfPlan the child operator of cubeOp by reference;
13)             return nullOp;
14)         }
15)     }
16)     return NULL ;
17) }
```

Figure 28: Pseudo-code for Rule 7.7

10.2 Statistics Retrieval

An accurate cost estimation mechanism depends considerably on accurate statistics. As described by [21], since the OLAP-XML federation is based on data sources from the web, a high degree of autonomy must be expected for XML components, while OLAP components may or may not provide access to cost information. *Query probing*[27] is a commonly used technique where spe-

cial queries, called *probing queries* are used to determine statistics. In this paper, we assume the OLAP component can provide basic statistical information through system catalog while the XML component has no cost information available and the only way is to use *probing queries*.

The following paragraphs describes the retrieval of the statistics that are listed in Section 9.1 and 9.5.1.

OVERHEAD is the initialization cost of the temporary and OLAP component. It is measured beforehand and then stored as a constant into the program. The idea of getting this information is to send a query that needs almost no evaluation and transfer time, that is, no data is actually returned. The duration time of this querying is measured by a Java program since the time the query is sent via a database connection until the query result is returned.

To get this constant, a group of queries were used, one of which is shown below,

```
SELECT * FROM TestingTable
```

where *TestingTable* is empty. Therefore, the average processing time of those queries is taken as *OVERHEAD*.

DataWriteRate is the rate with which data can be written back into physical storage devices. Like *OVERHEAD*, it is a constant stored in the program. Following the ideas of Equation 9.2.1 on Page 40, the following queries are designed.

```
SELECT * INTO tempdb..[#T1] FROM LineItem;
SELECT * INTO tempdb..[#T2] FROM LineItem;
       :
SELECT * INTO tempdb..[#T9] FROM LineItem;
SELECT * INTO tempdb..[#T10] FROM LineItem;
```

where, table names starting with "#T" are the temporary tables that are going to be built, and LineItem is a table of about 50MB of TPC-H data. The idea is to copy the data ten times into 10 different tables. Through observations, we have found that the first query takes much more time than others, while the other queries take approximately same processing time. For the first query, the processing time includes the overhead, the loading time and writing time. The data is assumed to be loaded into memory after the first query, thus the processing time for the rest queries is the writing time, that is: $\frac{Size(LineItem)}{DataWriteRate}$. Therefore, the average processing time of the rest 9 queries is computed, and then *DataWriteRate* can be retrieved.

DataReadRate is the rate with which data can be loaded into memory from physical storage. The rate can be retrieved from Equation 9.2.1 after *DataReadRate* is estimated.

AttrSize is a function which returns the size of the given attributes. Data types of the attributes can be found in system catalogs. For numeric data types, e.g. integer and float, each attribute takes

the space of the length of its data type in bytes, whereas an attribute of character data types uses the average of the actual minimal length and the actual maximal length. Then the size of the given attributes is the sum of these lengths.

Cardinality is a function which returns the number of tuples in a given base table. This information can be retrieved from system catalogs.

Selectivity is a function which returns the fraction of the table that satisfies a given predicate. This can be estimated using standard methods, i.e. by assuming a uniform distribution, and by considering cardinality, minimum and maximum values of the involved attributes [9]. E.g., if $\theta = "Order \leq k"$, where k is constant, the selectivity of θ can be estimated by $Selectivity(\theta, T) = \frac{k - min(Order)}{max(Order) - min(Order)}$. In previous work [1], the selectivity for complex predicates are hard to approximate, and thus can be assumed to be 10%. In the current version of the query engine, the selectivity is set to a constant value of 10%.

NumDistinct is a function which returns the number of distinct values for the given attributes $attrs$. The assumption that the data is uniformly distributed is made, which is widely adopted in many cost estimation activities [9, 23, 17]. Under the data distribution assumptions, we can mathematically approximate the number of distinct values using the following method in [10]. `if` r `elements are chosen uniformly and at random from a set of n elements, the expected number of distinct elements obtained is` $n - n(1 - 1/n)^r$, where r is the number of tuples in the table where $attrs$ resides, and let n_i represent the number of distinct values of the $i\text{-}th$ attribute in $attrs$ which can be retrieved from the system catalog, thus $n = n_1 \times \ldots \times n_{|attrs|}$. But in the current version, $NumDistinct(attrs)$ is approximated using $Cardinality(T) \times 10\%$, where T is a table having the attributes given in $attrs$.

FactSize is a function which returns the size of a fact containing only given measures. Let $meas$ be the set of given measures, *FactSize* returns $AttrSize(meas)$, that is, the sum of the lengths of the data types of $meas$.

Size is a function which returns the size of a given table or a cube. For a table T, the size is the product of the *Cardinality*(T) and *AttrSize*($attrs$) where $attrs$ is the set of attributes in a row of T. For a cube C, the function returns the size of the fact table, [21] pointed out that the size of the fact table is the size of the facts containing all the measures, therefore the product of the cardinality of the fact table and the sum of the data lengths of all the measures is the size of cube C.

RecordLoadRate is the rate with which records are loaded into the query engine through database connections, e.g. ActiveX Data Objects (ADO) connections [6]. To get this rate, a query which

returns the whole table was sent. The duration between when the query was sent and the result was returned is the processing time, which also can be expressed by: $OVERHEAD + \frac{Size(T)}{DateReadRate} + \frac{Size(T)}{RecordLoadRate}$. Therefore *RecordLoadRate* can be computed.

RollUpFraction is a function which returns the fraction to which cube C is reduced in size, when it is rolled up to the given levels $\mathcal{L}$. As pointed out by [21], a method proposed by [23] can be used to estimate the size of multidimensional aggregates, which is based on the assumption that facts are distributed uniformly in the cube and does not consider the actual contents of the cube. Following the proposal in [21], *RollUpFraction* can be estimated by the following formula. $RollUpFraction(\mathcal{L}, C) = \frac{NumDistinct(\mathcal{L})}{r}$, where r is the number of facts in C, i.e. $r = \frac{Size(C)}{FactSize(\{M_1,\ldots,M_m\})}$, $\{M_1, \ldots, M_m\}$ represents the set of all the measures.

InlineRate is the rate with which the new predicate string is generated. The optimizer first loads the required data, then uses the data to build the new predicate string. The rate is approximated by the cost formula $t_{Inlining,\theta}$ on Page 47 for the first situation where $\theta = L/Link/xp\ po\ K$. The inlining time is measured from the time when the query is sent to load the required values until the new predicate string is generated. Since other statistics in the formula can be approximated using the above methods, $InlineRate$ can then be computed.

FactTransRate is the rate with which the OLAP facts are transfered from the cube to temporary database. First a query to retrieve the base cube C is sent. The transferring time is measured from when the records are returned until the fact table is built in the temporary database. As the size of the cube can be approximated using the method described above, the $FactTransRate$ can be computed by formula: $FactTransRate = \frac{Size(C)}{t_{OLAP,Trans}}$.

XMLOVERHEAD is the initialization cost of processing an XPath query. The overhead may include everything from parsing the entire XML document to parsing only the query. To get this information, a probing query was used, which specifies the first child node of the context node until the leaf node is reached, e.g. "/Nation/Nation[1]/NationName[1]" was sent to the XML component to get the first "NationName" node of the first element "Nation" in the XML document "Nations.xml". The query takes negligible evaluation time and returns a node of very small size. The processing time is measured when the probing query is sent until the query result is returned. The average processing time of a number of queries is taken as the $XMLOVERHEAD$.

NodeSize is the average size in bytes of the nodes pointed to by $path_E$. Several probing queries are used as described below:

$AbsXP_x[RelXP_x = v_i]$	$for\ NatLink : \mathcal{P}(L \times X \times AbsXP_x \times RelXP_x \times Alias)$
$AbsXP_{x_{v_i}}$	$for\ EnLink : \mathcal{P}(L_1 \times X \times AbsXP_x)$

where, v_i is the $i\text{-}th$ random value, $i \in \{1, \ldots, n\}$. The probing queries are 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 the queries for the given type of link. The nodes are then analyzed locally to find $NodeSize$ for the remaining elements including the nodes pointed to by $path_E$.

Cardinality is the total number of elements pointed to by $path_E$. Using the same queries as for $NodeSize$, information about cardinality can be found in the nodes returned by the probing queries.

XMLLoadRate is the rate with which XML data can be loaded into the temporary component. The load rate is approximated beforehand and stored as a constant value in the program. The way to get the information is to use the formula $XMLLoadRate(x) = \frac{NodeSize(xp) \times Cardinality(xp)}{t_\tau - XMLOVERHEAD}$. A query "/Nations/Nation/NationName" was used and the processing time t_τ is measured from when the query was sent until the result was returned. As the other statistical information can be approximated using the methods introduced above, $XMLLoadRate$ can then be computed.

10.3 Cost Estimation

Cost estimation is repeatedly invoked in plan space pruning. Given a plan tree (partial or complete), the cost estimation function is applied iteratively on each operator and derives the statistical summary (i.e. cardinality) of the output data and the cost of the operation. Once the cost of the sub-node operators is estimated, the cost of a partial plan is generated using the cost formula for SQL_{XM} queries given in Section 9.

Each operator has three attributes, *OpCost*, *Cardinality* and *OverAllCost*. *OpCost* is the cost of the operation that the operator performs given the input from its sub-operator(s). *Cardinality* is the cardinality of the resulting table. *OverAllCost* is the cost of the plan rooted at the current operator. In Figure 29, *generateCost* is a function that produces the cost of the given plan where the root operator is *op*. Before the cost of the root operator is estimated, the same function is invoked on each of the sub-node operators. When the functions have all returned, $opCost$ can be approximated by function *generateOpCost* using the cost and cardinality information from its sub-nodes. The cost of the plan is estimated using the cost model in Section 9, which is the sum of the current operator's cost and the maximal sub-plan cost. By this way, the estimated cost is propagated through a plan tree in the bottom-up direction and finally becomes the cost of the plan when the root is reached. Note that the input argument for each function is a reference to *op*.

```
double generateCost(Operator op)
1)  {
2)      maxSubCost = 0;
3)      for each child operator childOp of op
4)      {
5)          tempCost = generateCost(childOp);
6)          if tempCost > maxSubCost
7)              maxSubCost = tempCost;
8)      }
9)      generateOpCost(op);
10)     op.OverAllCost = op.OpCost + maxSubCost;
11)     return op.OverAllCost;
12)  }
```

Figure 29: Pseudo-code for Cost Estimation

```
generateOpCost(Operator op)
1)  {
2)      estimate op.OpCost using the cost formulas in Section 9;
3)      estimate op.Cardinality using the formulas for cardinality in Section 9;
4)  }
```

Figure 30: Pseudo-code for Cost Estimation

10.4 Component Query Construction

A plan is evaluated by sending component queries to the respective components. In this section, we introduce how the component queries are constructed from the operators.

Constructing OLAP Queries The cube operators below the fact transfer in a plan can be refined to a general form: $\sigma_{Cube[\theta_1]}(\Pi_{Cube[\mathcal{L}]<F(M)>}(\sigma_{Cube[\theta_2]}(\mathcal{F})))$, where the parameter predicates are all in conjunctive normal form. From this an OLAP component query is constructed:

SELECT	F(M), $\mathcal{L}$
FROM	N
WHERE	θ_2
GROUP BY	$\mathcal{L}$
HAVING	θ_1

where N is the cube's name.

As described in Section 4.2, a dimension transfer results in a table of (bottom level value, non-bottom level value) pairs being added to the temporary cube. This is acheived by sending a simple OLAP query down to the OLAP component. For a dimension transfer operator $\omega_{[\perp_z, l_z]}(\mathcal{F})$, the following query is used,

SELECT $\perp_z, l_z$
INTO $T_{\omega_{[\perp_z, l_z]}}$
FROM N

where N is the cube name, $T_{\omega_{[\perp_z, l_z]}}$ is a table name returned by a *Naming function* $TableID : Op \longrightarrow T_{Op}$, which returns an unique name for the temporary table generated by Op. In the following descriptions, T_{Op} refers to the name generated by $TableID(Op)$.

Constructing The SQL Query The SQL query constructed for a federation selection $\sigma_{Fed[\theta]}(F)$ based on the fact table F of the temporary cube C is:

$Q_{\sigma_\theta}(F) =$ SELECT F.*
FROM F INNER JOIN $T_{\omega_{[\perp_i, l_i]}}$ ON $F.\perp_i = T_{\omega_{[\perp_i, l_i]}}.\perp_i$
... INNER JOIN $T\omega_{[\perp_j, l_j]}$ ON $F.\perp_j = T_{\omega_{[\perp_j, l_j]}}.\perp_j$
LEFT OUTER JOIN $T_{\delta_{[l_m/Link_u/xp_x]}}$
ON $F.\perp_m = T_{\delta_{[l_m/Link_u/xp_x]}}.\perp_m$
... LEFT OUTER JOIN $T_{\delta_{[l_n/Link_v/xp_y]}}$
ON $F.\perp_n = T_{\delta_{[l_n/Link_v/xp_y]}}.\perp_n$
WHERE θ

where, there exist dimension levels and decoration levels refered by the predicates but not present in F, that is, $\{l_i, \ldots, l_j\} \in RLevels(\theta) \wedge l_i \neq \perp_i \wedge \ldots \wedge l_j \neq \perp_j \wedge \{l_m/Link_u/xp_x, \ldots, l_n/Link_v/xp_y\} \in RLExprs(\theta)$, and F is joined with the temporary tables generated by dimension transfer and decoration operators, that is, $l_i(\perp_i) \rightsquigarrow \omega_{[\perp_i, l_i]} \wedge \ldots \wedge l_j(\perp_j) \rightsquigarrow \omega_{[\perp_j, l_j]}$ and $l_m/Link_u/xp_x \rightsquigarrow \delta_{[l_m/Link_u/xp_x]} \wedge \ldots \wedge l_n/Link_v/xp_y \rightsquigarrow \delta_{[l_n/Link_v/xp_y]}$. $RLevels(\theta)$ is the function that returns the referenced dimension levels. $RLExprs(\theta)$ returns the level expressions referenced by θ.

Example 10.1 For the federation selection, $\sigma_{Fed[Nation='DK']}(\mathcal{F})$, let F be the table in Figure 6, T1 in Figure 7 is the table generated by $\omega_{[Supplier, Nation]}(\mathcal{F})$, that is, T1=$TableID(\omega_{[Supplier, Nation]}(\mathcal{F}))$, the generated SQL query is:

```
SELECT  F.*
FROM    F INNER JOIN T1
        ON F.Supplier=T1.Supplier
WHERE   Nation='DK'
```

Similarly, the SQL query constructed for a federation generalized projection $\Pi_{Fed[\mathcal{L}]<F(M)>}(F)$ is:

$$
\begin{array}{lll}
Q_{\Pi_{Fed[\mathcal{L}]<F(M)>}}(F) = & \text{SELECT} & F(M), \mathcal{L} \\
& \text{FROM} & F \text{ INNER JOIN } T_{\omega_{[\perp_i,l_i]}} \text{ ON } F.\perp_i = T_{\omega_{[\perp_i,l_i]}}.\perp_i \\
& & \ldots \text{ INNER JOIN } T_{\omega_{[\perp_j,l_j]}} \text{ ON } F.\perp_j = T_{\omega_{[\perp_j,l_j]}}.\perp_j \\
& & \text{LEFT OUTER JOIN } T_{\delta_{[l_m/Link_u/xp_x]}} \\
& & \text{ON } F.\perp_m = T_{\delta_{[l_m/Link_u/xp_x]}}.\perp_m \\
& & \ldots \text{ LEFT OUTER JOIN } T_{\delta_{[l_n/Link_v/xp_y]}} \\
& & \text{ON } F.\perp_n = T_{\delta_{[l_n/Link_v/xp_y]}}.\perp_n \\
& \text{GROUP BY} & \mathcal{L}
\end{array}
$$

where, there exist dimension levels and decoration levels present in the select arguments but not in F, that is, $\{l_i, \ldots, l_j, l_m/Link_u/xp_x, \ldots, l_n/Link_v/xp_y\} \in \mathcal{L} \wedge l_i \neq \perp_i \wedge \ldots \wedge l_j \neq \perp_j \wedge l_i \notin Attrs(F) \wedge \ldots \wedge l_j \notin Attrs(F)$, and F is joined with the temporary tables generated by dimension transfer and decoration operators, that is, $l_i(\perp_i) \rightsquigarrow \omega_{[\perp_i,l_i]} \wedge \ldots \wedge l_j(\perp_j) \rightsquigarrow \omega_{[\perp_j,l_j]}$ and $l_m/Link_u/xp_x \rightsquigarrow \delta_{[l_m/Link_u/xp_x]} \wedge \ldots \wedge l_n/Link_v/xp_y \rightsquigarrow \delta_{[l_n/Link_v/xp_y]}$.

Example 10.2 For the federation selection, $\sigma_{Fed[Nation='DK']}(\mathcal{F})$, let F be the table in Figure 6, T1 in Figure 7 is the table generated by $\omega_{[Supplier,Nation]}(\mathcal{F})$, that is, T1=$TableID(\omega_{[Supplier,Nation]}(\mathcal{F}))$, the generated SQL query is:

```
SELECT   SUM(Quantity), SUM(ExtPrice), T1.Nation
INTO     Temptable
FROM     T1 INNER JOIN T2
         ON T1.Supplier=T2.Supplier
GROUP BY T1.Nation
```

A complete SQL query can be constructed for the federation operators on the path from the fact transfer to the root where the possible *inlining* and null operators on the path are not concerned as they do not change the fact table of the temporary cube. Let Op_{bottom} be the operator above the fact-transfer, Op_{top} be the root operator, the query construction starts from Op_{bottom} by repeatedly nesting the SQL query for the current operator in the FROM clause of the SQL query for the above operator, until Op_{top} is reached. For example, the SQL query for the following plan

$$\Pi_{Fed[\mathcal{L}]<F(M)>}(\sigma_{Fed[\theta_1]}(\Pi_{Fed[\mathcal{L}]<F(M)>}(\sigma_{Fed[\theta_2]}(\phi(\mathcal{F}), \Upsilon_4), \Upsilon_3), \Upsilon_2), \Upsilon_1)$$

is $Q_{\Pi_{Fed[]<F(M)>}}(Q_{\sigma_{Fed[\theta_1]}}(Q_{\Pi_{Fed[\mathcal{L}]<F(M)>}}(Q_{\sigma_{Fed[\theta_2]}}(F))))$.

Usually the nested SQL query can be flattened during its construction. For example, the WHERE clause and the HAVING clause can be added to the query generated for a generalized projection, e.g. if a federation generalized projection is found below a federation selection, then the HAVING clause is instead generated for the selection operator and attached to the end of the query for the lower projection. Repeatedly joined tables are removed when two SELECT statements are combined into

one, e.g. if a federation selection is found below a generalized projection and has the same or subset of the joined tables required by the projection, then one SQL query is produced for the two federation operators. For the query $Q_{\Pi_{Fed[]<F(M)>}}(Q_{\sigma_{Fed[\theta_1]}}(Q_{\Pi_{Fed[\mathcal{L}]<F(M)>}}(Q_{\sigma_{Fed[\theta_2]}}(F))))$, the flattened query is

SELECT $F(M), \mathcal{L}$
FROM (SELECT $F(M), \mathcal{L}$
FROM F INNER JOIN $T_{\omega_{\perp_i,l_i}}$ ON $F.\perp_i = T_{\omega_{[\perp_i,l_i]}}.\perp_i$
... INNER JOIN $T_{\omega_{[\perp_j,l_j]}}$ ON $F.\perp_j = T_{\omega_{[\perp_j,l_j]}}.\perp_j$
LEFT OUTER JOIN $T_{\delta_{[l_m/Link_u/xp_x]}}$
ON $F.\perp_m = T_{\delta_{[l_m/Link_u/xp_x]}}.\perp_m$
... LEFT OUTER JOIN $TableID(\delta_{[l_n/Link_v/xp_y]}$
ON $F.\perp_n = T_{\delta_{[l_n/Link_v/xp_y]}}.\perp_n)$
WHERE θ_2
GROUP BY $\mathcal{L}$
HAVING θ_1)
GROUP BY $\mathcal{L}$

where, there exist dimension levels and decoration levels such that $\{l_i, \ldots, l_j\} \in \mathcal{L} \cup RLevels(\theta_1) \cup RLevels(\theta_2) \wedge l_i \neq \perp_i \wedge \ldots \wedge l_j \neq \perp_j \wedge l_i \notin Attrs(F) \wedge \ldots \wedge l_j \notin Attrs(F) \wedge \{l_m/Link_u/xp_x, \ldots, l_n/Link_v/xp_y\} \in \mathcal{L} \cup RLExprs(\theta_1) \cup RLExprs(theta_2)$, and F is joined with the temporary tables generated by dimension transfer and decoration operators, that is, $l_i(\perp_i) \rightsquigarrow \omega_{[\perp_i,l_i]} \wedge \ldots \wedge l_j(\perp_j) \rightsquigarrow \omega_{[\perp_j,l_j]}$ and $l_m/Link_u/xp_x \rightsquigarrow \delta_{[l_m/Link_u/xp_x]} \wedge \ldots \wedge l_n/Link_v/xp_y \rightsquigarrow \delta_{[l_n/Link_v/xp_y]}$. Notice that the outer SELECT statement aggregates the temporary fact table again since the plan fragment for this example is not optimized. The outer aggregation will be removed if all the applicable transformation rules are applied.

As described in Section 4.4, each decoration operator results in a temporary dimension table of (bottom dimension value, decoration value) pairs added to the temporary cube. The table is built by joining two tables which are loaded by a dimension transfer and an XML transfer. For a decoration operator $\delta_{[l_z/Link/xp]}(\mathcal{F})$, the following query is constructed.

SELECT DISTINCT l_z, l_{xp}
INTO $T_{\delta_{[l_z/Link/xp]}}$
FROM $T_{\omega_{[\perp_z,l_z]}}$ INNER JOIN $T_{\tau_{[l_z/Link/xp]}}$
ON $T_{\omega_{[\perp_z,l_z]}}.l_z = T_{\tau_{[l_z/Link/xp]}}.l_z$

Example 10.3 For the decoration operator $\delta_{[Nation/NLink/Population]}(\mathcal{F})$, T1 in Figure 7 is loaded by $\omega_{[Supplier,Nation]}(\mathcal{F})$ and table T2 in Figure 8 is loaded by $\tau_{[Nation/NLink/Population]}(\mathcal{F})$, then the external dimension table is created by the query below.

```
SELECT   DISTINCT Supplier, Population
FROM     T1 INNER JOIN T2
         ON T1.Nation = T2.Nation
```

Constructing XML Queries As defined in Section 4.3, each XML-transfer operator results in a table of (dimension value, decoration value) pairs being added to the temporary cube. The XML queries that are needed to fetch the relevant data from the XML components, depend on the type of link used in the level expression. For each tuple (e, URI, locator) in the enumerated links, the dimension value (e) is already present, to get a decoration value pointed at a level expression $l/Link/xp$, the following query is used, $locator/xp$. For natural links, a node is identified for each dimension value by the locator part of the link. Thus, an XML query must relate the node identified by the locator to the node identified by the user specified XPath expression. The following query is used, $base/locator\ |/xp$, which returns the pairs of a node containing the identifying value and the node specified by the XPath expression xp in the context of $base$. For example, given a natural link, NLink = ("Nation", "nations.xml", "/Nations/Nation/", "NationName") and a level expression, "Nation/NLink/Population", the XPath query is "/Nations/Nation/NationName | /Population".

11 Performance Studies

We have performed a set of experiments to evaluate the performance of the optimized query engine. The experiments were carried out on the same hardware and software environment as in [26].

The plan selected by the query planner is the cheapest in the plan space. To see how this plan actually makes the performance boosts, we evaluated the initial plan, the selected plan and the intermediate plans in between on our new query engine.

The user query is:

```
SELECT     SUM(Quantity),SUM(ExtPrice),Nation(Supplier),Brand(Part),
           LineStatus(LineNumber), Nation/NLink/Population
FROM       TData
WHERE      Nation/NLink/Population>90000000
GROUP BY   Nation(Supplier),Brand(Part),LineStatus(LineNumber)
```

The experimental result is shown in Figure 31. Y axis represents the logarithm of the evaluation time in milliseconds, while the six numbers on X axis represent plans that are in the process of optimization. Plan 1 is the initial plan, Plan 2 to 5 represent the five intermediate plans we captured during the plan optimization process. Plan 7 is the final result of the optimization, which is the selected plan with the least cost estimated by the query planner.

The initial plan is shown in Figure 32 After Rule 7.10 is applied, an inlining operator is created above the σ surrounded by a dotted line box in Figure 32, the predicate is marked and references to the level expression will be inlined at evaluation time. In the figure, we can see, for both databases, the evaluation time of Plan 2 is larger than the time of Plan 1. That is because the predicate is

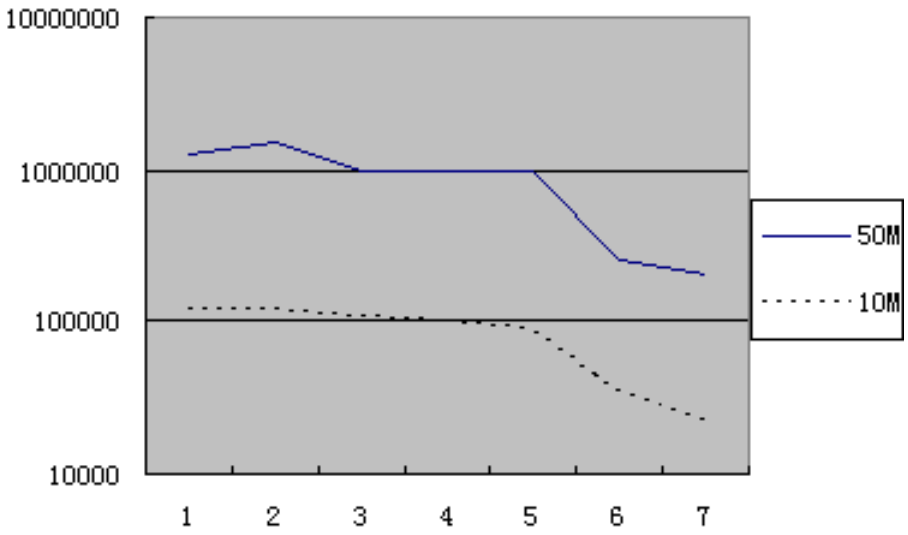


Figure 31: The Evaluation Time of Plans During Optimization

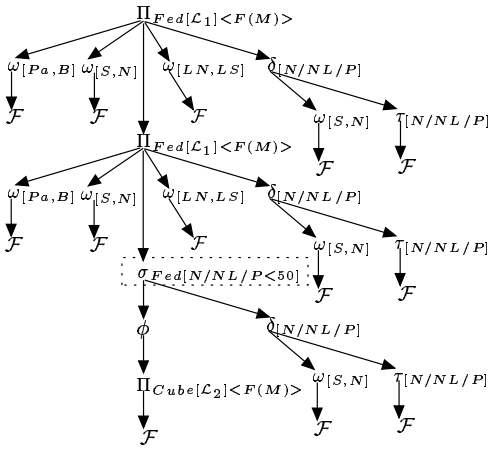


Figure 32: Plan 1 - The Initial Plan

rewritten, but the selection and decoration are still processed in the temporary area. Plan 2 is shown in Figure 33

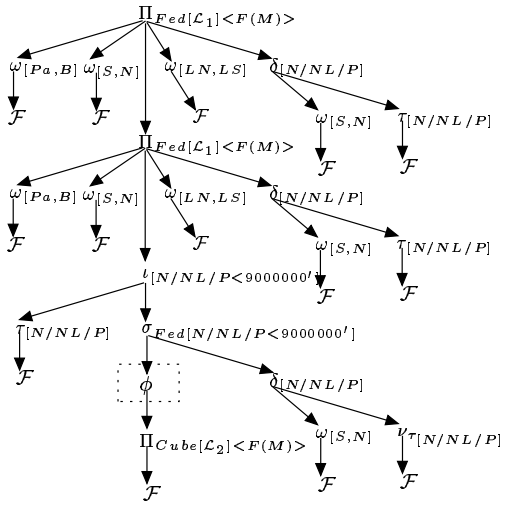


Figure 33: Plan 2

After Rule 7.7 is applied, the federation selection is pushed below the fact transfer surrounded by a box in Figure 33. In Figure 31, the evaluation time drops for both databases, that is because the selection is evaluated in the OLAP component which is much faster than the federation selection, the transfered data from the OLAP cube to the temporary cube is also reduced. We can get a significantly performance boost if the cube selection has very selective predicates. The inlining of decoration data enables the push down of federation selections, thus an inlining is always good when the predicate transformation is not very expensive and the new predicates are selective. Plan 3 is shown in Figure 34.

After Rule 7.11 is applied twice, the inlining operator surrounded by a dotted line box in Figure 34 is moved to the top, two federation GPs are pushed down. Nothing more is changed, thus we can see from the experimental results, the evaluation time remains almost the same. Plan 4 is shown in Figure 35.

After Rule 7.8 is applied, the null operator surrounded by a box in Figure 35 is combined with the above federation GP. The former decoration and dimension transfer of the null operator are redundant and thus then removed, which reduces the evaluation time to some extent. Plan 5 is shown in Figure 36.

After Rule 7.3 is applied, a new cube projection is created below the fact transfer in Figure 36. Rule 7.17 switches the cube selection and the cube projection that retrieves all the measures and bottom levels. Rule 7.14 then removes the redundant cube projection below the newly created cube projection. The cube projection surrounded by a dotted line box in Figure 36 will be removed. Plan

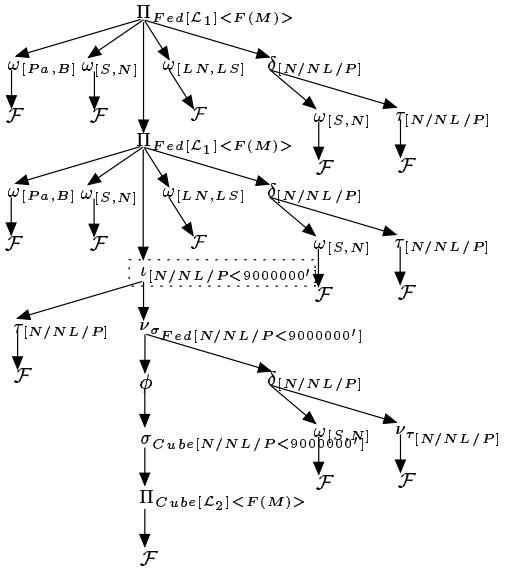


Figure 34: Plan 3

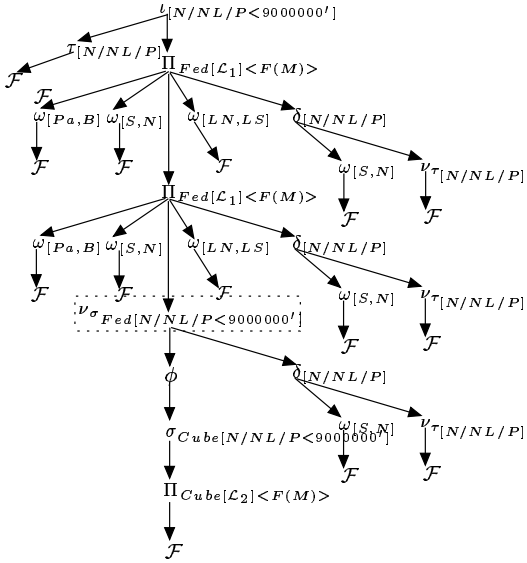


Figure 35: Plan 4

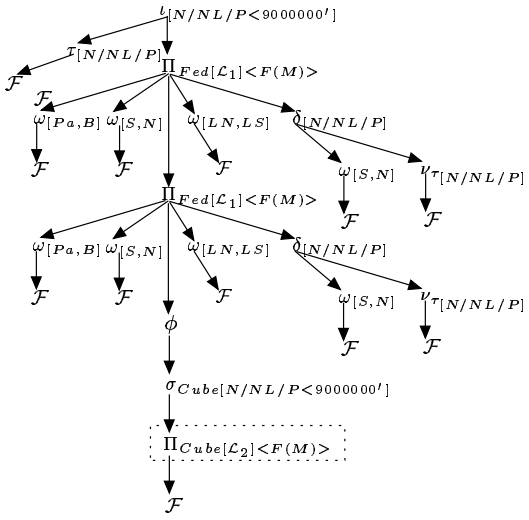


Figure 36: Plan 5

6 is shown in Figure 37. The evaluation time is reduced a lot for Plan 6 since the cube generalized projection aggregates the cube.

The last rule applied is Rule 7.4, the duplicate dimension transfers and decorations are removed, only one federation GP is kept. Thus evaluation time is reduced for the removal of repeated operations. The federation GP with a surrounded box in Figure 37 will be removed. Plan 7 is shown in Figure 38.

Figure 39(a) and 39(b) show the exact evaluation time for the seven plans processed on the 50M and the 10M cubes. The unit of Y axis in each figure is second. In both test environments, the selected execution plan (Plan 7) is evaluated almost 6 times faster than the initial plan.

From the above experiments, we can see the optimizations really help in reducing the evaluation time. However, not every applied transformation rule can improve the performance, which on the contrary may grow worse, e.g. Plan 2 introduces a new operator which brings extra processing. But the plan generated afterwards uses Plan 2 as a stepping-stone and gives a good result on performance. Therefore, we can see that the plan rewriting is sort of "team" work of the transformation rules and a good optimizer should not be "near-sighted". The *inlining* technique saves much more evaluation time when the transformed predicate is more selective. Plan 3 for the 50M cube has a better performance than the same plan evaluated on the 10M cube. The transformed predicates are the same, but the data selected in the larger cube takes a larger proportion than that of the smaller one. Much more evaluation time will be saved if the base cube is aggregated as Plan 6 shows. Elimination of the duplicate operators also saves evaluation time, but as the DBMS uses caching to reduce the cost of repeated operations, the performance gain for duplicate elimination is not very

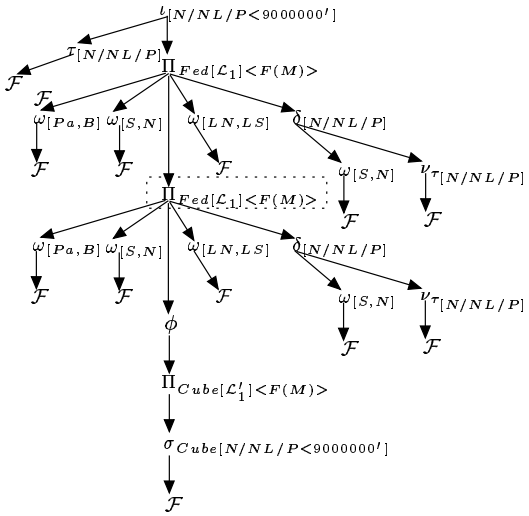


Figure 37: Plan 6

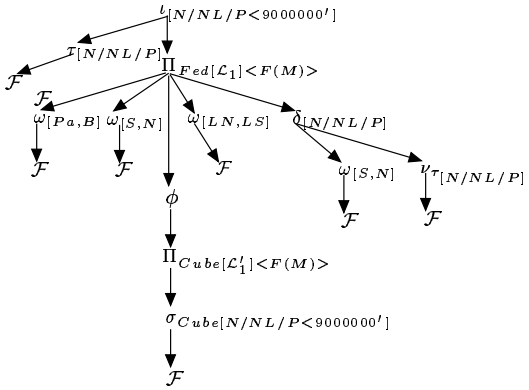
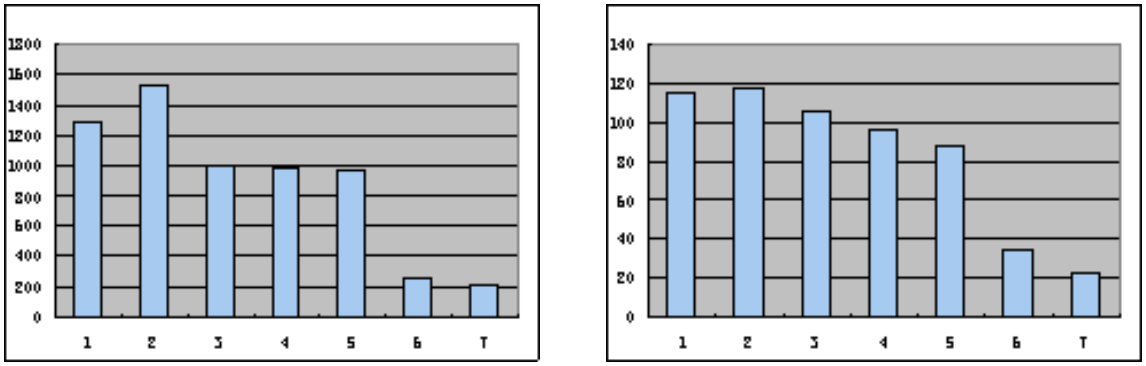


Figure 38: Plan 7



(a) 50M Cube

(b) 10M Cube

Figure 39: Evaluation Time of Plans

significant.

12 Conclusion

OLAP systems today have a common problem in physically integrating fast changing data. Thus, the logical integration that enables the external data available in XML databases virtually as part of the local OLAP database becomes a desirable solution. Motivated by this, previous work [19, 20, 21, 26] designed a OLAP-XML query engine that supports the logical federation of OLAP and XML databases. OLAP data now can be decorated, sliced by XML data. Measures can be aggregated over the grouped XML data. A prototype of OLAP-XML query engine was developed and basic queries can be processed.

In this paper, extensions have been made to the logical OLAP-XML federation system. The physical algebra of SQL_{XM} are extended with new operators and new operator semantics. An SQL_{XM} optimizer that consists of a plan space generator and a best plan selection. Rule-based plan generation is achieved by applying transformation rules. Cost-based plan selection is achieved by using cost models to estimate cost of operators and the size of their output. A prototype query engine with optimizations is implemented. Experiments have been carried out to show the efficiency of optimizations, indicating that as enhanced by practical optimization techniques, the logical OLAP-XML federation is now much more applicable and becomes a practical alternative to physical federation.

In future work, a more mature query engine should be developed. More advanced or alternative implementation algorithms could be developed, e.g. indexes can be built for temporary tables, the SQL queries we create for the federation operations can be tuned before they are sent to DBMS and etc.. The cost models could be revised to give more accurate estimates. More efficient transforma-

tion rules could be developed, and other optimization techniques, e.g. caching/pre-fetching, could be integrated into the query engine to produce more promising querying performance.

References

[1] José A. Blakeley, William J. McKenna, and Goetz Graefe. Experiences building the open oodb query optimizer. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 287-296, 1993.

[2] James Clark and Steve DeRose, XML Path Language (XPath), http://www.w3.org/TR/xpath, 2003. Current as of January 09, 2003.

[3] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7-18, 1994.

[4] Jens Clausen. Branch and Bound Algorithms - Principles and Examples. http://www.imm.dtu.dk/ jha/TSPtext.pdf, 1999. Current as of Jue 01, 2003.

[5] IBM Corporation. Datajoiner. http://www-4.ibm.com/software/data/datajoiner, 2003. Current as of January 08, 2003.

[6] Microsoft Corporation, Microsoft Developer Network, http://msdn.microsoft.com, 2003. Current as of June 01, 2003.

[7] Oracle Corporation. Gateways. http://www.oracle.com/gateways, 2003. Current as of January 08, 2003.

[8] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. Query optimization in a heterogeneous dbms. In *Proceedings of 18th International Conference on Very Large Data Bases*, pages 277-C291, 1992.

[9] Rames Elmasri, Shamkant B. Navathe. *Fundamentals of Database Systems*, third Edition, 2000.

[10] W. Feller. *An Introduction to Probability Theory and Its Applications*, Vol. I, John Wiley and Sons, pp 241, 1957.

[11] Goetz Graefe and W.J.McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of IEEE ICDE*, Vienna, Austria, pp. 209-218 (1993).

[12] Junmin Gu, Torben Bach Pedersen, Arie Shoshani: OLAP++: Powerful and Easy-to-Use Federations of OLAP and Object Databases. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 599-602, 2000.

[13] Roy Goldman and Jennifer Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 285-296, 2000.

[14] Joseph M. Hellerstein, Michael Stonebraker, and Rick Caccia. Independent, open enterprise data integration. *IEEE Data Engineering Bulletin*, 22(1):43-49, 1999.

[15] Yannis E. Ioannidis. Query Optimization. *ACM Computing Surveys*, 28(1), pages 121-123, 1996.

[16] Tirthankar Lahiri, Serge Abiteboul, and Jennifer Widom. Ozone - integrating semistructured and structured data. In *Proceedings of the 8th International Workshop on Database Programming Languages*, 1999.

[17] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas and Eugene J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of ACM SIGMOD*, pages 294-305, 1996.

[18] Torben Bach Pedersen and Christian S. Jensen. Multidimensional data modeling for complex data. In *Proceedings of the 15th International Conference on Data Engineering*, pages 336-345, 1999.

[19] Dennis Pedersen, Karsten Riis, and Torben Bach Pedersen: XML-Extended OLAP Querying. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 195-206, 2002.

[20] Dennis Pedersen, Karsten Riis, and Torben Bach Pedersen: Cost Modeling and Estimation for OLAP-XML Federations. In *Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery*, pages 245-254, 2002.

[21] Dennis Pedersen, Karsten Riis, and Torben Bach Pedersen: Query optimization for OLAP-XML federations. In *Proceedings of the Fifth International Workshop on Data Warehousing and OLAP*, pages 195-206, 2002.

[22] Torben Bach Pedersen, Arie Shoshani, Junmin Gu, and Christian S. Jensen: Extending OLAP Querying to External Object Databases. In *Proceedings of the 2000 ACM CIKM International Conference on Information and Knowledge Management*, pages 405-413, 2000.

[23] Amit Shukla, Prasad Deshpande, Jeffrey F. Naughton, and Karthikeyan Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proceedings of 22nd International Conference on Very Large Data Bases*, pages 522-C531, 1996.

[24] Transaction Processing Performance Council. TPC-H. http://www.tpc.org/tpch, 2003. Current as of January 08, 2003.

[25] W3C. Extensible markup language (xml) 1.0 (second edition). http://www.w3.org/TR/REC-xml, October 2000. Current as of June 15, 2001.

[26] Xuepeng Yin, Towards an OLAP-XML Query Engine, KDE3 Semester report, Department of Computer Science, Aalborg University, Jan. 2003.

[27] Qiang Zhu and Perke Larson. Global query processing and optimization in the cords multidatabase system. In *Proceedings of the 9th ISCA International Conference on Parallel and Distributed Computing Systems*, pages 640-646, 1996.