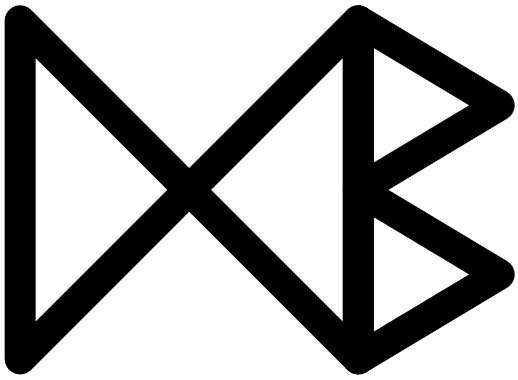


Integrating XML Data In The TARGIT OLAP System

Dennis Pedersen, Jesper Pedersen, and Torben Bach Pedersen

May 5, 2004

TR-8

A DB Technical Report

Title	Integrating XML Data In The TARGIT OLAP System
Author(s)	Dennis Pedersen, Jesper Pedersen, and Torben Bach Pedersen
Publication History	March 2004, *Proceedings of the Nineteenth International Conference on Data Engineering*, pp. 778–781 May 2004, a DB Technical Report

For additional information, see the DB TECH REPORTS homepage: ⟨www.cs.auc.dk/DBTR⟩.

The DB TECH REPORTS icon is made from two letters in an early version of the Rune alphabet, which was used by the Vikings, among others. Runes have angular shapes and lack horizontal lines because the primary storage medium was wood, although they may also be found on jewelry, tools, and weapons. Runes were perceived as having magic, hidden powers. The first letter in the logo is "Dagaz," the rune for day or daylight and the phonetic equivalent of "d." Its meanings include happiness, activity, and satisfaction. The second letter is "Berkano," which is associated with the birch tree. Its divinatory meanings include health, new beginnings, growth, plenty, and clearance. It is associated with Idun, goddess of Spring, and with fertility. It is the phonetic equivalent of "b."

Abstract

This paper presents the results of industrial work on the logical integration of OLAP and XML data sources, carried out in cooperation between TARGIT, a Danish OLAP client vendor, and Aalborg University. A prototype has been developed that allows XML data stored outside the OLAP system to be used as dimensions and measures in the OLAP system in the same way as ordinary dimensions and measures. This provides a powerful and flexible way to handle unexpected or short-term data requirements as well as rapidly changing data. Compared to earlier work, this paper presents several major extensions that resulted from TARGIT's requirements. These include the ability to use XML data as measures, as well as a novel *multigranular* data model and query language that formalizes and extends the TARGIT data model and query language.

1 Introduction

This paper presents the results of industrial work on the logical integration of OLAP and XML data sources, carried out in cooperation between TARGIT [12], a Danish OLAP client vendor, and Aalborg University. TARGIT's practical experience with the OLAP industry and their expectations for its future development [7] has provided a valuable input to earlier work [8] resulting in several important theoretical extensions and refinements as well as the development of a prototype closely integrated with TARGIT's current product.

The most significant theoretical extension is the ability to define logical *measures* from external XML data, e.g., a new measure could be added containing sales forecasts, with the underlying data stored in an XML document on the company intranet. This would allow queries such as "Show sales and sales forecasts by month and city" even though the forecasts are not physically stored in the cube. Additionally, the ability to use external XML data as *dimensions* [8] have been adapted to the TARGIT framework. Another major contribution is the *formalization and extension* of the TARGIT system's internal data model and query language. Unlike other multidimensional data models and query languages, the ones presented here are *multigranular*, i.e., they allow fact data to have *varying granularity* over a dimension. This paper also documents the prototype development effort, which includes adapting and refining the previously developed techniques for query processing and optimization to fit TARGIT's recommendations and platform. Although we briefly consider user interface aspects, the focus is on the logical level and on implementation issues. Experiments with the prototype indicate an acceptable overhead by using external dimensions and measures for a large range of useful queries. We believe these contributions to be novel and interesting to both the database research and industry communities.

There has been much previous work on data integration, e.g. on integrating relational data [4], object-oriented data [10], semi-structured data [1], and a combination of relational and semi-structured data [6]. However, none of this handles the advanced issues related to OLAP systems, e.g., dimensions with hierarchies, automatic aggregation, and the problems related to correct aggregation. This is also true for the XML query language XQuery [15], and for nD-SQL [2], which considers the federation of relational sources providing basic OLAP functionality. As mentioned, the most related work is [8], which is mostly theoretical, considers only external *dimensions*, and describes a *loosely coupled* federation [11]. In contrast, this paper focuses on implementation issues, allows external *measures*, uses the TARGIT data model and query language, and describes a *tightly coupled* federation where the user is not required to know implementation details.

The rest of the paper is organized as follows. Section 2 gives an overview of the TARGIT system. Section 3 defines the data models and query languages for OLAP and XML data sources. The OLAP data model is extended with external dimensions and measures in Section 4. The architecture of the extensions to the TARGIT system is described in Section 5. Section 6 describes query processing and optimization. Section 7 concludes and points to future work.

2 The TARGIT System

The main part of the TARGIT system is an OLAP client, which is known primarily for its ease of use. As can be seen in Figure 1, it allows the browsing of multidimensional data using a wide variety of charts, maps, and tables. These objects are interconnected such that an action in one object is reflected in the other objects. For example, by clicking a country on a map, the other objects are automatically drilled down such that they display values for that country. Although the client is aimed at non-technical users, it also covers more advanced functionality such as data mining, user defined measures, and report building. In addition to the client tool, the TARGIT system includes a server that ensures uniform access to different kinds of data sources, and an administrator tool for configuring the server. The general architecture of the TARGIT system is shown in Figure 2(a). Whenever the user performs an operation in the client, such as drilling down in a dimension, the client issues a query to the server. This query is expressed in TARGIT's own multidimensional query language TSQL, which is described in detail later. The query evaluation depends on the type of data source. Two different types of sources can be accessed through the server: Relational tables and Microsoft Analysis Services cubes. When using relational tables, a star or snowflake schema is built from a set of tables in the administrator tool; otherwise the cube is typically constructed in Microsoft's Cube Editor. All details must be handled explicitly by the server when using a set of relational tables, whereas query processing is much simpler for cubes, where the TSQL query is simply translated to a single MDX query [3]. Thus, the server consists of two almost independent parts corresponding to the two types of sources.

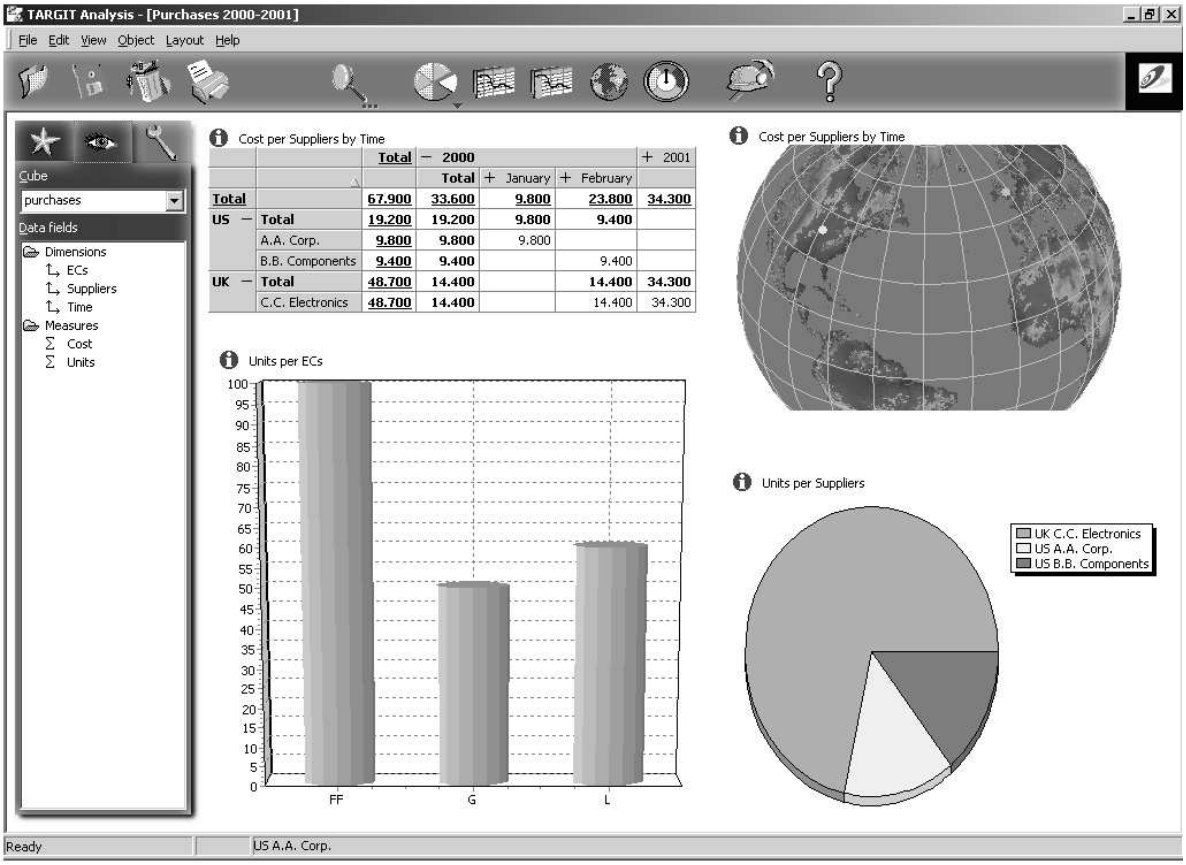
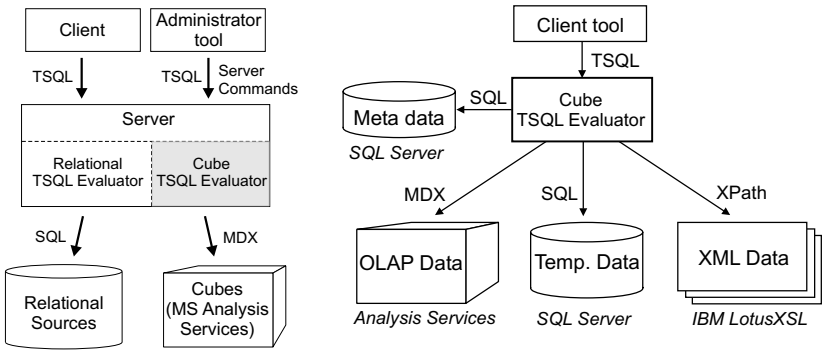


Figure 1: TARGIT Screenshot



(a) (b)

Figure 2: Current/future TARGIT architecture

The federation system presented in this paper extends the server with the ability to handle external dimensions and measures without having to integrate the data physically. The current prototype implementation presented in this paper is limited to Analysis Services cubes (the shaded area in Figure 2(a)) but extending this to cover relational sources is straightforward. Extending the server comprises two main tasks, namely extending the current handling of cube sources with the ability to *use external dimensions and measures* defined from XML sources, and *extending the TSQL language* with constructs to define these external cube parts. The client GUI should also be extended to support this.

3 Data Models and Query Languages

We now present a formalization of the TARGIT OLAP data model and query language, as well as the data model and query language used for accessing XML sources. The models are used to explain how external XML data is integrated into OLAP cubes and to discuss the theoretical implications of the integration. The OLAP model builds on the one in [9], capturing common multidimensional terms such as facts, dimensions, and hierarchies. However, the TARGIT OLAP model and query language has one feature that distinguishes it from almost all other multidimensional data models: the fact data resulting from a query may be of *varying granularity*, i.e., both lower and higher level aggregate values may be present in the same result set, and the occurence of lower/higher level aggregate values may even vary over a dimension. The latter feature distinguishes the TARGIT model and language from the well-known SQL CUBE operator. An algebra is defined over the OLAP model to describe how multidimensional cubes are queried, and this algebra is then used to define the exact semantics of the TSQL query language.

The examples are based on a case study concerning B2B portals, where a cube tracks the cost and number of units for *purchases* made by customer companies. The cube has three dimensions: Electronic Component (EC), Time, and Supplier. External data is found in an XML document that tracks component, unit, and manufacturer information. The XML document has the following nesting of elements and attributes: Components(Supplier(Class(Component(Manufacturer <@MCode> UnitPrice Description)))). The details of the case study are described in [8].

3.1 The OLAP Data Model

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

Definition 1 (Dimension) A *dimension* D_i is a two-tuple (L_{D_i}, E_{D_i}), where L_{D_i} is a poset of levels and E_{D_i} is a poset of dimension values.

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

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

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

For each level L we assume a function $\text{Ancestors}_L : \text{Values}(L) \times LS_i \mapsto \mathcal{P}(D_i)$, which given a dimension value in L and a level in LS_i returns the value's ancestors in the level. That is, $\text{Ancestors}_L(e, L') = \{e' \in L' | e \sqsubseteq_{D_i} e'\}$. Similarly, a function $\text{Children}_L : \text{Values}(L) \mapsto \mathcal{P}(D_i)$ returns the dimension values that are children of a given dimension value, i.e. $\text{Children}_L(e) = \{e' | e \sqsubseteq_{D_i} e' \wedge \nexists e''(e \sqsubseteq_{D_i} e'' \wedge e'' \sqsubseteq_{D_i} e')\}$. □

The intuition behind the partial order $\sqsubseteq_i$ of levels is that given two levels $L_{i1}, L_{i2} \in D_i$ we say that $L_{i1} \sqsubseteq_i L_{i2}$ if elements in L_{i2} can be said to contain the elements in L_{i1}. For example, $Day \sqsubseteq Year$ because years contain days. Similarly, we say that $e_1 \sqsubseteq e_2$ if e_1 is logically contained in e_2 and $L_{ij} \sqsubseteq_i L_{ik}$ for $e_1 \in L_{ij}$ and $e_2 \in L_{ik}$ and $e_1 \neq e_2$. For example, the day 01.21.2000 is contained in the year 2000. Note that a lower-level value may roll up to more than one higher-level value, i.e., dimensions may be *non-strict* [8]. Non-strict hierarchies may cause incorrect aggregation as data values may be double-counted. To prevent this, we distinguish between three different *aggregation types* of data: c, data that may not be aggregated, ϕ, data that may be averaged but not added, and Σ, data that may also be added. Thus, we have the following ordering of these types: $c \subset \phi \subset \Sigma$. Considering only the standard SQL functions, we have that $\Sigma = \{\text{SUM, AVG, MAX, MIN, COUNT}\}$, $\phi = \{\text{AVG, MAX, MIN, COUNT}\}$, and $c = \emptyset$. A function $\text{AggType} : \{M_1, \ldots, M_m\} \times D \mapsto \{\Sigma, \phi, c\}$ returns the aggregation type of a measure M_j when aggregated in a dimension $D_i \in D$. Aggregation types are used both to prohibit semantically incorrect aggregation, and to prevent aggregation when irregular hierarchies may lead to incorrect results.

Example 3.1 The case cube has a Time dimension, an ECs dimension and a Suppliers dimension. Letting Sup denote $Suppliers$ the Suppliers dimension consists of the levels $LS_{Sup} = \{\top_{Sup}, Country, Supplier\}$, which are ordered as follows: $\sqsubseteq_{Sup} = \{(Supplier, \top_{Sup}), (Country, \top_{Sup}), (Supplier, Country)\}$. Thus, the poset of levels is $L_{D_{Sup}} = (LS_{Sup}, \sqsubseteq_{Sup}, \top_{Sup}, Supplier)$.
The poset of dimension values is $E_{D_{Sup}} = (\{\top_{D_{Sup}}, US, UK, A.A., B.B., C.C.\}, \sqsubseteq_{D_{Sup}})$, where $\sqsubseteq_{D_{Sup}} = \{(US, \top_{D_{Sup}}), (UK, \top_{D_{Sup}}), (A.A., \top_{D_{Sup}}), (B.B., \top_{D_{Sup}}), (C.C., \top_{D_{Sup}}), (A.A., US), (B.B., US), (C.C., UK)\}$. Hence, the Suppliers dimension is given by: $D_{Sup} = (L_{D_{Sup}}, E_{D_{Sup}})$. □

Definition 2 (Cube and Fact Table) An *n-dimensional cube* is a three-tuple $C = (N, D, F)$ consisting of a cube name N, a non-empty set of dimensions $D = \{D_1, \ldots, D_n\}$ and a *fact table* $F(D_1, \ldots, D_n, M_1, \ldots, M_m)$ which is a relation containing one attribute for each dimension D_i and one attribute for each type of numerical value M_j, called a *measure*. Thus, $F = \{(e_1, \ldots, e_n, v_1, \ldots, v_m) | (e_1, \ldots, e_n) \in L_1 \times \cdots \times L_n (L_i \in D_i) \wedge (v_1, \ldots, v_m) \in M \subseteq T_1 \times \cdots \times T_m\}$,

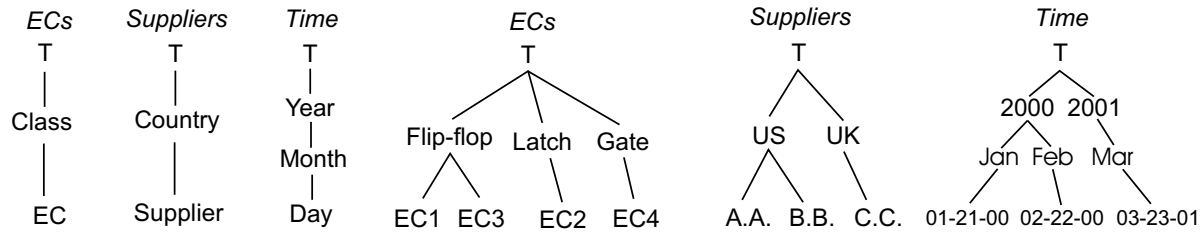


Figure 3: Purchases cube schema+instance

where $n \geq 1$, $m \geq 1$, and T_j is the domain value for the j'th measure. A function Level : $D_i \mapsto LS_i$ identifies the level to which a dimension value in the fact table belongs.

The measure domains T_j all contain the special NULL value, which denotes that no value exists for a particular combination of dimension values. A tuple in F, where at least one measure value exists, is called a *fact*. Each measure M_j is associated with a *default aggregate function* $f_j : \mathcal{P}(T_j) \mapsto T_j$, where the input is a multi-set. Aggregate functions ignore NULL values as in SQL. □

Intuitively, a tuple in F captures the measured values associated with one combination of dimension values. That is, there is one tuple in F for each possible combination of the dimension values. This is, of course, just logically, in a physical implementation only the non-empty tuples would be stored. Since the combination of dimension values are not required to be from the bottom levels, there can also be higher-level aggregates in a fact table.

Example 3.2 From the Purchases cube in the case study we can construct a three-dimensional cube with the cube name $Purchases$, the dimensions, levels, and ordering of dimension values as depicted in Figure 3. An example fact table, with higher-level aggregate values, is seen in Table 1. □

3.2 The Cube Algebra

In this section we present an algebra for the OLAP data model presented in Section 3.1. Two operators are defined that will be used to give the semantics of a TSQL query: generalized projection and selection. Each of these operators takes a cube as input and produces a cube as output. The expressive power of the algebra is limited to what is needed for defining the semantics of the TSQL language.

The generalized projection (GP) operator Π aggregates measure values in a cube such that the resulting fact table contains a specified set of dimension value combinations. Any measures or dimensions that are not specified, are not part of the resulting cube. The operator's parameters are a set of dimension values for each dimension in the result and a set of measures.

Example 3.3 Calculate the cost and number of units for combinations of the Supplier level and the Class level as well as all higher-level aggregates:

$\Pi_{[\{\top,\mathrm{US},\mathrm{UK},\mathrm{A.A.},\mathrm{B.B.},\mathrm{C.C.}\},\{\top,\mathrm{FF},\mathrm{G},\mathrm{L}\}]<\mathrm{SUM}(\mathrm{Cost}),\mathrm{SUM}(\mathrm{Units})>}(\mathrm{Purchases}) = \mathrm{Purchases}'$

The fact table of the new Purchases′ cube is shown in Table 1. In addition, the Time dimension and the EC level has been removed from the Purchases cube. □

The bottom-most dimension values specified for each dimension defines the bottom levels of the resulting cube. Levels below these are not part of the result and neither are the measures not specified in the argument list. If no dimension values are listed for a dimension, this dimension is aggregated to the top

Cost	Units	Suppliers	ECs
9800	3000	A.A. Supplier	FF Class
9400	3000	B.B. Supplier	FF Class
14400	4000	C.C. Supplier	FF Class
16700	5000	C.C. Supplier	G Class
17600	6000	C.C. Supplier	L Class
19200	6000	US Country	FF Class
14400	4000	UK Country	FF Class
16700	5000	UK Country	G Class
17600	6000	UK Country	L Class
33600	10000	$\top$ $\top_{\text{Sup}}$	FF Class
16700	5000	$\top$ $\top_{\text{Sup}}$	G Class
17600	6000	$\top$ $\top_{\text{Sup}}$	L Class
9800	3000	A.A. Supplier	$\top$ $\top_{\text{EC}}$
9400	3000	B.B. Supplier	$\top$ $\top_{\text{EC}}$
48700	15000	C.C. Supplier	$\top$ $\top_{\text{EC}}$
19200	6000	US Country	$\top$ $\top_{\text{EC}}$
48700	15000	UK Country	$\top$ $\top_{\text{EC}}$
67900	21000	$\top$ $\top_{\text{Sup}}$	$\top$ $\top_{\text{EC}}$

Table 1: Result fact table for Example 3.3

level and removed from the cube. Each new measure value is calculated by applying the given aggregate function to the corresponding measure value for all tuples in the old fact table that contain bottom values that roll up to the dimension values of the new fact table.

Notice that rolling up to a higher level may result in duplicated facts if the hierarchy is non-strict, i.e. if a dimension value has more than one parent. To ensure safe aggregation in case of non-strict hierarchies we explicitly check for this in each dimension [8, 9]. If a roll-up along some dimension duplicates facts *at the bottom level*, we disallow further aggregation along that dimension by setting the aggregation type to c. Since the higher-level aggregates are always calculated from the bottom level values, all aggregate values will be correct as long as the bottom level contains no duplicated values. If it does, further aggregation would have been disallowed when the duplicates were produced.

Formally, we define:

Definition 3 (Generalized projection) Let $C = (N, D, F)$ be a cube. Then the generalized projection operator is defined as: $\Pi_{[\{e_{i_1 1},\ldots,e_{i_1 n_1}\},\ldots,\{e_{i_k 1},\ldots,e_{i_k n_k}\}]<f_{j_1}(M_{j_1}),\ldots,f_{j_l}(M_{j_l})>}(C) = (N', D', F')$, where each $\{e_{i_h 1}, \ldots, e_{i_h n_h}\}$ is a set of dimension values specifying the aggregations for dimension $i_h \in \{1, \ldots, n\}$. The measures $\{M_{j_1}, \ldots, M_{j_l}\} \subseteq \{M_1, \ldots, M_m\}$ are kept in the cube. $f_{j_1}, \ldots, f_{j_l}$ are the given aggregate functions for the specified measures, such that each of them are valid in all dimensions in which aggregation occurs, i.e. $\forall D_{i_h} \in \{D'_{i_h} \in D | \exists e \in \{e_{i_h 1}, \ldots, e_{i_h n_h}\}(e \notin \bot_{i_h})\}(\forall f_{j_g} \in \{f_{j_1}, \ldots, f_{j_l}\}(f_{j_g} \in \text{AggType}(M_{j_g}, D_{i_h})))$.

The resulting cube is given by: $N' = N$ and $D' = \{D'_{i_1}, \ldots, D'_{i_k}\}$, where $D'_{i_h} = (L'_{D_{i_h}}, E'_{D_{i_h}})$ for $h = 1, \ldots, k$.

Let $\{\mathcal{L}_{i_1}, \ldots, \mathcal{L}_{i_k}\}$ be the set of levels that contain the bottommost dimension values mentioned for each of the dimensions, i.e. the set $\{\mathcal{L}_{i_h} \in D_{i_h} | \forall e \in (\mathcal{L}_{i_h} \cap \{e_{i_h 1}, \ldots, e_{i_h n_h}\})(\nexists e' \in \{e_{i_h 1}, \ldots, e_{i_h n_h}\}(e' \sqsubset_{D_{i_h}} e)\}$. Then the new poset of levels in the remaining dimensions is $L'_{D_{i_h}} = (LS'_{i_h}, \sqsubseteq'_{i_h}, \top_{i_h}, \mathcal{L}_{i_h})$, where $LS'_{i_h} = \{L_{i_h p} \in LS_{i_h} | \mathcal{L}_{i_h} \sqsubseteq_{i_h} L_{i_h p}\}$, and $\sqsubseteq'_{i_h} = \sqsubseteq_{i_h}|_{LS'_{i_h}}$. Moreover, $E'_{D_{i_h}} = (\bigcup_p L_{i_h p}, \sqsubseteq_{D_{i_h}}|_{\bigcup_p L_{i_h p}})$.

The new fact table is given by: $F' = \{(e'_{i_1}, \ldots, e'_{i_k}, v'_{j_1}, \ldots, v'_{j_l}) | e'_{i_h} \in \{e_{i_h 1}, \ldots, e_{i_h n_h}\} \wedge v_{j_h} = f_{M_{j_h}}(\{v | (e_{\bot_1}, \ldots, e_{\bot_n}, v) \in M_{j_h} \wedge (e'_{\bot_{i_1}}, \ldots, e'_{\bot_{i_k}}) \in \text{Ancestors}_{\bot_{i_1}}(e_{\bot_{i_1}}, \mathcal{L}_{i_1}) \times \cdots \times \text{Ancestors}_{\bot_{i_k}}(e_{\bot_{i_k}}, \mathcal{L}_{i_k})\})\}$.

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

As a short-hand, we will use $\Pi_{[\{L_1,\ldots,L_k\}]<M_1,\ldots,M_l>}$ to mean $\Pi_{[\{\text{Values}(L_1),\ldots,\text{Values}(L_k)\}]<M_1,\ldots,M_l>}$.

The *selection* operator σ is used to slice the cube so that it contains only facts that satisfy a given predicate. The predicates we consider here are restricted to the subset of the usual SQL operators that are allowed in the TSQL language, namely =,<,>,<>, and IN which can be combined with AND. (OR is not used in TSQL predicates.) These operators can be used to compare constants and levels. A selection only affects the tuples in the fact table. Hence, selection returns a cube with the same fact type and the same set of dimensions. All tuples for which the predicate does not hold are removed, i.e. their measures are set to NULL. For any tuples that are computed from other tuples the measures are recalculated to reflect the lower-level changes. Formally, we define the selection operator as follows:

Definition 4 (Selection operator) Let p be a predicate over the set of levels $\{L_1,\ldots,L_k\}$ and measures $M_1,\ldots,M_m$. Selection on a cube $C = (N, D, F)$ is $\sigma_{[p]}(C) = (N', D', F')$, where $N' = N$, $D' = D$ and $F' = \{t'_1,\ldots,t'_l\}$. If $t_i = (e_1,\ldots,e_n,v_1,\ldots,v_m) \in F$ then

$$t'_i = \begin{cases} (e_1,\ldots,e_n,\text{NULL},\ldots,\text{NULL}) & \text{if } p(t_i) = \mathit{false} \\ (e_1,\ldots,e_n,v'_1,\ldots,v'_m) & \\ \quad \text{if } \exists e_i \in \{e_1,\ldots,e_n\} & \\ \quad (\exists(e''_1,\ldots,e''_n,v''_1,\ldots,v''_m) & \\ \quad\quad \in F(e''_i \sqsubset e_i)) & \\ t_i & \text{otherwise.} \end{cases}$$

where each $v'_j = f_{M_j}(\{v|(e_{\perp_1},\ldots,e_{\perp_n},v) \in M_j \wedge (e_1,\ldots,e_n) \in \text{Ancestors}_{\perp_1}(e_{\perp_1},\text{Level}(e_1)) \times \cdots \times \text{Ancestors}_{\perp_n}(e_{\perp_n},\text{Level}(e_n))\})$ □

3.3 The TSQL Query Language

In this section we define the semantics of TARGIT's TSQL language in terms of the data model and algebra defined above[1].

A TSQL query consists of a SELECT-FROM-WHERE-DRILL DOWN construct. The SELECT clause lists the dimensions and measures that should be part of the result, while the FROM clause specifies the cube. The optional WHERE clause is used to slice the cube by specifying a predicate over the dimension values as described in Section 3.2. DRILL DOWN lists the set of dimension values that should be present in the result using the *DimensionValue*.CHILDREN notation, which means the set of all dimension values that are children of *DimensionValue*.

Example 3.4 An example TSQL query is given below.

```
SELECT      [Suppliers], [ECs], [Units], [Cost]
FROM        [Purchases]
WHERE       [ECs].[EC] IN ('EC1', 'EC3', 'EC4')
DRILL DOWN  ([Suppliers].[US].CHILDREN,
            [Suppliers].[UK].CHILDREN)
```
□

[1]The syntax presented in this paper has been modified slightly for improved readability. Also, we focus on the primary querying abilities of the language, although it has other more specific purposes, such as retrieving cube metadata.

As mentioned, sub-predicates in the WHERE clause can only be connected by AND. This is because in the client tool predicates are constructed by adding primitive, i.e. non-conjunctive and non-disjunctive, selection predicates to a *criteria list* which must all be satisfied. The DRILL DOWN clause implicitly contains the top level and its immediate children for all dimensions. Thus, if the Suppliers dimension is mentioned in the SELECT clause, [Suppliers].[All] and [Suppliers].[All].CHILDREN are always part of the DRILL DOWN clause. Dimensions are treated as in MS Analysis Services, i.e. they cannot contain multiple hierarchies. Instead these are defined as distinct dimensions. Furthermore, a value's children can only be added to the DRILL DOWN set if the value is itself part of the DRILL DOWN set. That is, [Time].[2000].[January].CHILDREN is only allowed if [Time].[2000].CHILDREN is also listed. This makes sense from a user interface perspective because an element is added to the DRILL DOWN set by graphically "unfolding" a dimension value, and this is, of course, only possible if the value is already visible. As a shorthand, all dimension values in dimension D from the root and down to a level L can be specified as [D].[L]. Thus, the DRILL DOWN set in Example 3.4 could also be specified as [Suppliers].[Supplier].

A TSQL query can be expressed using a single selection and a single GP. The WHERE clause is evaluated first using the selection operator and then the aggregation is performed using the GP as prescribed by the DRILL DOWN clause including its implicit parts. The general form of a TSQL query is:

$$
\begin{array}{ll}
\text{SELECT} & D_1, \ldots, D_n, M_1, \ldots, M_m \\
\text{FROM} & C \\
\text{WHERE} & p(D_1, \ldots, D_n) \\
\text{DRILL DOWN} & (D_{1,1}.e_{1,1,1}.\ldots.e_{1,1,k_{11}}. \\
 & \text{CHILDREN}, \ldots, D_{1,l_1}.e_{1,l_1,1}.\ldots. \\
 & .e_{1,l_1,k_{1l_1}}.\text{CHILDREN} \\
 & \qquad\qquad \vdots \\
 & D_{n,1}.e_{n,1,1}.\ldots.e_{n,1,k_{n1}}. \\
 & \text{CHILDREN}, \ldots, D_{n,l_n}.e_{n,l_n,1}.\ldots. \\
 & .e_{n,l_n,k_{nl_n}}.\text{CHILDREN})
\end{array}
$$

where $l_i \geq 0$ and $k_{il_i} \geq 0$.

Each query on this form is equivalent to an algebra expression with the following general structure: $\Pi_{[\{\top,\mathcal{C}(\top),\mathcal{C}(e_{1,1,k_{11}}),\ldots,\mathcal{C}(e_{1,l_1,k_{1l_1}})\},\ldots,\{\top,\mathcal{C}(\top),\mathcal{C}(e_{n,1,k_{n1}}),\ldots,\mathcal{C}(e_{n,l_n,k_{nl_n}})\}]<M_1,\ldots,M_m>}(\sigma_{p(D_1,\ldots,D_n)}(C))$, where $\mathcal{C}$ represents the Children function stated in Definition 1.

3.4 XML Data Model and Query Language

The XPath language is used to refer to parts of XML documents. Although not a full blown query language, this language is sufficiently powerful for our purpose. XPath is also chosen because it has a compact syntax making it suitable for integration into another language. The XML data model underlying the XPath language views an XML document as a tree. Each node in the tree has one of the types: root, element, namespace, text, processing instruction, attribute, or comment. For more details about the XML data model and its use in our approach we refer to [13] and [8], respectively.

The basic syntax of an XPath expression resembles a Unix file path where a full path expression is given as a number of locations separated by a "/", e.g. location-step$_1$/. . . /location-step$_n$. The returned set of nodes can also be restricted by applying one or more *predicates* which supports the usual boolean, mathematical, and string operators.

Example 3.5 Select all ECs which are of the flip-flop class and are manufactured by either Johnson Components or by the manufacturer with code M33: /Components/Supplier/Class/Component[Manufacturer/

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

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

Definition 5 (XPath Expression) Let S be a set of nodes in an XML document. An XPath expression is a function XP : $S \mapsto \mathcal{P}(S)$ The set of all valid XPath expressions over an XML document x is called XP_x, while the subset of XP_x that are absolute XPath expressions is called $AbsXP_x$. That is, $AbsXP_x = \{xp \in XP_x | Dom(xp) = \text{Root}(x)\}$. $RelXP_x$ is the set of expressions in XP_x that are not in $AbsXP_x$. □

4 Integration with External XML Data

External data may be used as either dimension values or measure values in a cube. Using external data as dimension values allows the existing cube data to be grouped in new ways; For example, purchases may be grouped by the city in which the supplier is located, even if this information is not available in the cube but only in an external XML document. There are two general ways external data can be used as measure values: *Vertically*, by adding new measures to the cube, and *horizontally*, by adding additional facts of the same type to the cube. The former adds new columns to the fact table, while the latter adds new rows to it. As an example of the vertical case, a new measure based on transportation costs may be added, and for the horizontal case, additional purchase data may be specified for a new division that has not been represented in the cube before. As the focus here is on adding new *types* of information to existing cubes, horizontal extension is outside the scope of this paper.

This section extends the data model from the previous section with the notions of *external dimensions* and *external measures*. It also defines a flexible way to specify these based on external XML data. The algebra presented in the previous section is then extended with operations that merge external dimensions and measures with existing cubes, such that the new "virtual" cube can be queried as any other cube. Finally, these formal definitions are used to extend the TSQL language with facilities to handle external data. In the next section we look at how these queries can be evaluated in practice.

4.1 External Dimensions

An *external dimension* is defined to be a relationship between dimension values from a single level in a cube and nodes in an XML document. This is similar to, but more flexible than, the *property* concept used in many commercial OLAP systems, where a descriptive value is attached to each dimension value. The reason external dimensions are more flexible, is that they allow more than one external value per dimension value, which is usually not possible with properties. As we will see later, a cube can be "decorated" with the information represented by an external dimension, creating a new cube in which the external dimension values appear as a new dimension.

Definition 6 (External Dimension) An external dimension is a relation $XD \subseteq \{(e, s) | e \in L \wedge s \in S\}$ where L is a level in a cube and S is a set of nodes from an XML document. □

Example 4.1 The information needed to group purchases by the suppliers' cities as found in the Cities document is represented by the external dimension: SupplierCities = {("A.A.", "Los Angeles"), ("B.B.", "New York"), ("C.C.", "London, Manchester")} □

A very powerful way of specifying external dimensions is by giving an XPath expression in which the cube's level names can be used as variables. For example, the expression /SupplierCities/Supplier [SupplierName = $Supplier]/City identifies the cities in which a supplier is located given the supplier's name. This

situation, where some information in the cube can be used to identify the external values, is very common, since this information represents the *semantical relationship* between the two data sources. In the example, it is only natural that the suppliers' names are also present in the XML document, since without them the list of supplier cities would be useless. The level used to identify the external values is referred to as the *defining level*. However, the names that are used in the cube and in the XML document may not be of the same type, e.g. abbreviated names are used in the Purchases cube while the full names are used in the Cities document. To handle this, an *alias function* can be specified that maps between the two types of names. A more low-level way of specifying external dimensions that gives complete control of the identification of external values is presented in [8].

An important problem with the use of external values is *non-strictness*, i.e. what to do when there are more than one external node for each dimension value, as is the case for "C.C. Electronics" in the Cities.xml document. Several different semantics are appropriate in different situations, but the following have been found to cover most applications:

1. using all of the nodes as distinct values,
2. concatenating all the nodes into a single value, and
3. picking an arbitrary node.

The most flexible one is the first, since it allows grouping by the individual values, but it also presents some problems as discussed later. These problems do not exist when only a single external value is used for each dimension value. The general solution chosen here is to apply a user-defined multi-valued function to the set of nodes. This function may e.g. be one of those listed above.

Definition 7 (External Dimension Specification) Let $Levels$ be the set of all levels, $XMLDocuments$ be the set of all XML documents, $Aliases$ be the set of all alias functions, $XPathExpressions$ be the set of all XPath expressions, $NodeFunctions$ be the set of all multi-valued functions over nodes in an XML document, and $ExternalDimensions$ be the set of all external dimensions.

An *external dimension specification* is a function: $XDSpec : Levels \times XMLDocuments \times Aliases \times XPathExpressions \times NodeFunctions \rightarrow ExternalDimensions$

The resulting external dimension D_x is given by: $XDSpec(L, X, a, xp, f_{xp}) = \{(e, v) | e \in L \wedge v \in f_{xp}(xp(X, a(e)))\}$ where L is the defining level, X is an XML document, $a : L \rightarrow Alias$ is an alias function, which may be the identity function $Id : a(e) = e$, xp is an XPath expression over X using variable a, and $f_{xp} : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ is a node function. □

Although any node function can be used, most applications are covered by three functions corresponding to each of the three semantics discussed above: ALL, CONCAT, and ANY.

Example 4.2 The external dimension from Example 4.1 can be defined using the external dimension specification:

XDSpec("SupplierCities", Supplier, Cities.xml, {("A.A.", "A.A. Corp."), ("B.B.", "B.B. Components"), ("C.C.", "C.C. Electronics")}, /SupplierCities/Supplier[SupplierName=$Supplier]/City, CONCAT) □

Rather than redefining the existing operators to work also on external dimensions, we add a new operator δ^d that merges a cube and an external dimension into a new cube. The operator is called *decoration* since it "decorates" a dimension in a cube with additional information.

Example 4.3 The Purchases cube can be decorated with the SupplierCities dimension as follows:

$PurchasesCity = \delta^d_{[SupplierCities]}(Purchases)$

The result of the decoration is the Purchases cube plus the dimension shown in Figure 4. □

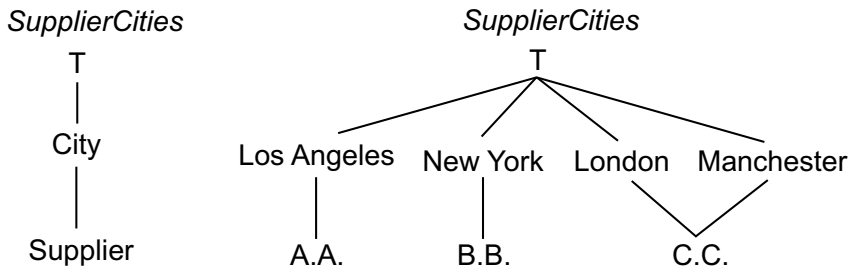


Figure 4: The new Supplier dimension

The formal definition of the decoration operation can be found in [8]. Since there can be more than one external value for each dimension value, the hierarchy may be *non-strict*, which means that facts may be duplicated when performing aggregation. As discussed in Section 3, this is handled by keeping track of the duplication and disallowing aggregations that may lead to false results.

4.2 External Measures

Extending cubes with new measure values in the vertical direction (in the following referred to as vertical extension or just extension) is more involved than with dimensions, since a measure depends on all the existing dimensions and not just one of them.

External measures are only allowed to contain values for a single combination of levels. Although it is possible to allow external measures at different levels, we advice against this, because of the semantical problems caused if there is inconsistency between the external values. For example, if the external data specifies non-strict hierarchies, there can be several ways to compute a higher-level aggregate, each resulting in different but equally (in)valid aggregate values. It is possible to handle this problem, e.g. by checking the consistency of the external data before using it, but that is outside the scope of this work. However, we do not require the external measure to be specified for the *bottom* combination of levels, which means that external values at different levels *can* be handled by creating multiple independent external measures. Since external measures need not be specified for the bottom level, the measure values may be missing for some combinations of levels. This is similar to the behavior of the drill-across operation for joining multiple cubes, where some measures may not be defined for all combinations of levels in the result.

We define an external measure to be a relation between dimension values for a combination of levels and a single external value:

Definition 8 (External Measure) An external measure is a relation $XM \subseteq \{(e_1, \ldots, e_n, v) | e_i \in L_i \wedge v \in MV \wedge \forall (e'_1, \ldots, e'_n, v') \in XM \backslash (e_1, \ldots, e_n, v)((e'_1, \ldots, e'_n) \neq (e_1, \ldots, e_n))\}$ where L_i is a level in the i'th dimension and MV is the domain for measure values. An external measure has a default aggregate function f_{XM} assigned to it. □

Example 4.4 The external measure OverheadExpenses can be created from the OverheadExpenses.xml document:

OverheadExpenses = {(⊤, "A.A.", "EC1", 200), (⊤, "A.A.", "EC4", 300), (⊤, "B.B.", "EC2", 1100), (⊤, "B.B.", "EC3", 800), (⊤, "C.C.", "EC1", 400), (⊤, "C.C.", "EC2", 600), (⊤, "C.C.", "EC4", 1000)}

Notice that the last value is aggregated. □

External measures are specified similarly to external dimensions. However, it is necessary to define values for a combination of levels instead of just for one level. If there are more than one value for each combination of dimension values, they must be transformed to a single value, typically by aggregating them or by selecting one of the values.

Definition 9 (External Measure Specification) In addition to the domains used in the definition of external dimension specifications, let $AggregateFunctions$ be a set of aggregate functions, $LevelLists$ be the set of all combinations of levels, $AliasLists$ be the set of all lists of alias functions, and $ExternalMeasures$ be the set of all external measures.

An external measure specification is a function: $XMSpec : LevelLists \times XMLDocuments \times AliasLists \times XPathExpressions \times NodeFunctions \times AggregateFunctions \rightarrow ExternalMeasures$.

Let $\mathcal{L} = (L_1, \ldots, L_n)$ be a list of levels such that for dimensions not used to identify the external values, the top level is specified, $A = (a_1, \ldots, a_n)$ be a list of the corresponding alias functions, X be an XML document, and xp be an XPath expression over X using variable names defined by A. Also, let f_{xp} and f_{M_x} be the node function and the default aggregate function, respectively. Then the resulting external measure M_x is given by: $XMSpec(\mathcal{L}, X, A, xp, f_{xp}, f_{M_x}) = \{(e_1, \ldots, e_n, v) | e_i \in L_i \wedge v \in MV \wedge v = \text{SingleValue}(X, A, xp, f_{xp})\}$ where $\text{SingleValue}(X, A, xp, f_{xp}) = f_{xp}(xp(X, a_{i_1}(e_{i_1}), \ldots, a_{i_k}(e_{i_k})))$if $xp(X, a_{i_1}(e_{i_1}), \ldots, a_{i_k}(e_{i_k})) \neq \emptyset, k \leq n$ and NULL, otherwise. □

Which node functions to use depends on the particular application, but either SUM or ANY makes sense in most situations. The aggregate functions used here are the standard SQL aggregate functions SUM, MIN, MAX, COUNT, and AVG. Notice that all levels not part of the identifying XPath expression must be specified as the top level. This is also similar to the behavior of the drill-across operation.

Example 4.5 The external measure in Example 4.4 can be specified from the OverheadExpenses.xml document as follows: OverheadExpenses = XMSpec(($\top_{Time}$, Supplier, EC), Overhead.xml, (Id, Id, Id), /OverheadExpenses/Supplier [@Name=$Supplier]/EC[@Name=$EC], SUM, SUM) □

The actual extension of a cube with the new measure is performed by the *vertical extension* operator δ^m. When extending a cube with an external measure, only the existing facts are extended. That is, no new facts are created. This is necessary to avoid creating facts that are incorrect with respect to the real world. For example, when adding a new measure "transportation cost" to the Purchases cube, the measure may be taken from an XML document with general information about transportation costs between major cities in the world. If all the external measure values were treated as new facts, a large number of new purchases that never actually happened would be created in the cube.

Formally, vertical extension is defined as:

Definition 10 (Vertical Extension) Vertical extension δ^m of a cube $C = (N, D, F)$ is defined as: $\delta^m_{[\text{XM}]}(C) = C'$ where $C' = (N, D, F')$. The new fact table is given by:
$F' = \{(e_1, \ldots, e_n, v_1, \ldots, v_m, v_x) | (e_1, \ldots, e_n, v_1, \ldots, v_m) \in F (\exists v_i \in \{v_1, \ldots, v_m\}(v_i \neq \text{NULL})) \wedge v_x = \text{Val}(e_1, \ldots, e_n)\}$ where $\text{Val}(e_1, \ldots, e_n) =$

$$\begin{cases} v_{ext} & \text{if} \exists (e_1, \ldots, e_n, v_{ext}) \in \text{XM} \\ f_{XM}(\{v\}) & \text{if} \nexists (e_1, \ldots, e_n, v_{ext}) \in \text{XM} \\ & \quad \wedge \{v | \exists (e'_1, \ldots, e'_n, v) \in XM (\forall e'_i \in \\ & \quad \{e'_1, \ldots, e'_n\} (e'_i \sqsubseteq e_i))\} \neq \emptyset \\ \text{NULL} & \text{otherwise.} \end{cases}$$

□

Vertical extension only affects the fact table. Intuitively each fact is extended with any external information that is available, either directly from the external measure or by calculating it from lower-level values in the external measure. If this is not possible, the measure value is NULL, denoting that it is missing.

Example 4.6 The result of extending Purchases″ with the OverheadExpenses measure (OE) is shown in Table 2. □

Cost	Units	OE	Suppliers	ECs
9800	3000	200	A.A. Supplier	FF Class
9400	3000	800	B.B. Supplier	FF Class
14400	4000	400	C.C. Supplier	FF Class
16700	5000	1000	C.C. Supplier	G Class
19200	6000	1000	US Country	FF Class
14400	4000	400	UK Country	FF Class
16700	5000	1000	UK Country	G Class
33600	10000	1400	$\top$ $\top_{\mathrm{Sup}}$	FF Class
16700	5000	1000	$\top$ $\top_{\mathrm{Sup}}$	G Class
9800	3000	200	A.A. Supplier	$\top$ $\top_{\mathrm{EC}}$
9400	3000	800	B.B. Supplier	$\top$ $\top_{\mathrm{EC}}$
31100	9000	1400	C.C. Supplier	$\top$ $\top_{\mathrm{EC}}$
19200	6000	1000	US Country	$\top$ $\top_{\mathrm{EC}}$
31100	9000	1400	UK Country	$\top$ $\top_{\mathrm{EC}}$
50300	15000	2400	$\top$ $\top_{\mathrm{Sup}}$	$\top$ $\top_{\mathrm{EC}}$

Table 2: Fact table external OE measure

4.3 Extensions to TSQL

Using the decoration and vertical extension operators, the selection and GP operators can be used as on ordinary cubes. However, a few changes to the TSQL language are needed to allow the definition of external dimensions and measures. The following changes are made to TSQL's SELECT construct: 1) external dimensions and measures can now be referenced in the SELECT clause, and 2) levels in external dimensions can be used in the WHERE clause. Notice that dimension values in external dimensions cannot be used in the DRILL DOWN clause. This is because external dimensions only have a single level below the top level (apart from the bottom level which is always identical to the bottom level of the dimension being decorated) and both of these levels are implicitly part of the DRILL DOWN clause.

Example 4.7 The following TSQL query includes the external dimension and measure defined above:

```
SELECT      [SupplierCities], [ECs], [Cost],
            [OverheadExpenses]
FROM        [Purchases]
WHERE       [SupplierCities].[City] IN
            ('Los Angeles', 'London')
DRILL DOWN  ([ECs].[FF].CHILDREN,
            [ECs].[L].CHILDREN)
```
□

In addition, two CREATE statements, and corresponding DROP statements, are defined below to provide a practical way to create and drop new dimensions and measures in a cube based on external XML data. A CREATE EXTERNAL DIMENSION statement specifies an external dimension, while the CREATE EXTERNAL MEASURE specifies an external measure. Of course, since the data is located externally, the data is not retrieved and the dimensions or measures are not created until a query is actually evaluated. The values specified in each of the statements correspond closely to the elements of the dimension and measures specifications. We will show the syntax by the two examples below.

Example 4.8 The TSQL equivalent of the specification in Example 4.2 is:

```
CREATE EXTERNAL DIMENSION SupplierCities
IN CUBE Purchases
FROM Cities.xml
IDENTIFIED BY /SupplierCities/Supplier[SupplierName
  =$Supplier]/City
USING Supplier AS $Supplier
WITH ALIASES SupplierNameMappings
WITH SEMANTICS CONCAT
```

Here, SupplierNameMappings refers to the name of a table containing supplier-alias pairs. □

Example 4.9 The external measure specification in Example 4.5 is equivalent to the following TSQL query:

```
CREATE EXTERNAL MEASURE OverheadExpenses
IN CUBE Purchases
FROM OverheadExpenses.xml
IDENTIFIED BY /OverheadExpenses/Supplier[@Name
  =$Supplier]/EC[@Name=$EC]
USING (Supplier AS $Supplier, EC AS $EC)
WITH SEMANTICS SUM
WITH AGGREGATE FUNCTION SUM
```

□

Before presenting the prototype system supporting these extensions to the TSQL language, we will briefly consider how the changes affect the user interface.

4.4 Extending the User Interface

The new facilities in the TSQL language are primarily intended to be used by database administrators and sophisticated users. For the average user, who will just *use* the external data, there will simply be more dimensions and measures to select from when building cube views. However, one of the key selling points of the TARGIT system is its short installation and configuration time and thus, the user interface is still of primary concern. Moreover, with the right user guidance (e.g. by the use of "wizards") it may be possible for the average user to add external data, at least with a reduced set of configuration options. However, this UI design challenge is outside the scope of this paper.

The following is a sketch of a usage situation for a database administrator using the administrator tool. 1) the user chooses a cube and selects either "Add external dimension" or "Add external measure" in the administrator tool; 2) the user is asked to identify an XML document. This can be done by typing in a URL or by "capturing" the URL from a regular web browser. It can also be selected from a predefined list of the company's external data sources or it may be provided by a third-party "data broker"; 3) the structure of the XML document is displayed graphically as a tree next to the structure of the cube. Any correspondences between XML and cube data that can be identified automatically (based on their names and possibly the data) are marked by links between the two graphical representations; 4) the user can now create new links by dragging XML nodes onto the cube representation. While doing this, the data is analyzed in the background to determine whether all the corresponding values in the cube and in the XML document can be derived automatically. If any values cannot be matched, the user is asked to specify aliases for these values; 5) the user is asked for the name of the external dimension or measure, the semantics, the default aggregate function, etc. Default values are suggested based on the data.

When an external dimension or measure has been specified, the user of the client tool can select these external data from a list in the same way ordinary dimensions and measures are used. Thus, only very few changes are needed in the client tool.

5 Architecture of the TARGIT Extension

Overall Architecture In current OLAP systems, including MS Analysis Services, which is used for the prototype, it is not possible to add new dimensions and measures without processing the cube. MS Analysis Services [3], allows so-called *changing dimensions* to be created, which do not require the cube to be fully processed, but a partial processing is still needed, making it infeasible to do at query time. Also, *non-strict* dimensions are not allowed, which is necessary to provide flexible handling of external data. This means that external dimensions and measures cannot be implemented simply by adding them physically to the cube at query time and, consequently, a different approach is taken here. The basic idea is to evaluate a TSQL query by constructing and evaluating an OLAP query and a set of XML queries separately, and then combining the results of these queries using a relational database. This has the additional advantage that any OLAP server can be used. In order to achieve acceptable performance with this approach, it is assumed that the temporary tables can be stored on the same machine as the OLAP database.

The overall architecture is shown in Figure 2(b) with an indication of which technologies are used in the prototype. The Cube TSQL Evaluator (referred to as TSQL Evaluator in the following), processes TSQL queries fed to it by the client tool by fetching data from the OLAP and XML components and combining it in a temporary component. A relational DBMS is used for the temporary component because the final result can easily be computed by joining the fact table resulting from the OLAP component query, and tables containing the external data. Another auxiliary component stores metadata used in the evaluation of a TSQL query such as specifications of the external dimensions and measures. Both the temporary and the metadata component uses the MS SQL Server DBMS.

TSQL Evaluator The architectural design of the TSQL Evaluator is shown in Figure 5. When a TSQL query is posed to the TSQL Evaluator it obtains an available *Request Thread* from a pool of threads and passes the query to it. The Request Thread uses the *Parser* to transform the query string into a query tree and passes this on to the *Query Evaluator*. Here, it is determined whether the *Cache Manager* can provide the requested result or a part of it. At this point, a number of asynchronous requests for external data not found in the cache are issued to the *Execution Engine*.

The next step is for the Query Evaluator to find an optimal evaluation plan, considering various optimization techniques as discussed in Section 6. When selecting the optimal evaluation plan the Query Evaluator makes use of the cost estimates provided by the *Global Cost Evaluator*. Based on the plan, the Execution Engine asynchronously evaluates an MDX query on the cube. All queries issued by the Execution Engine are performed by a *Component Interface*, which also handles any local optimization of the individual queries and maintains the cost information for its data source. If the result of an XML query becomes available before it was anticipated and the OLAP query has not yet been posed, the global plan is reconsidered. Upon completion of the component queries, the Execution Engine informs the Cache Manager about the intermediate results that have been added to the temporary component during execution of the TSQL query. These results can then be used by the Cache Manager until they expire.

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

6 Query Processing

Given the overall architecture presented above, the evaluation of a TSQL query is performed in these main steps: 1) *split* the TSQL query into a pure OLAP query and a set of XML queries; 2) evaluate these queries in parallel; 3) join the OLAP fact table and the tables containing external data as specified by the decoration

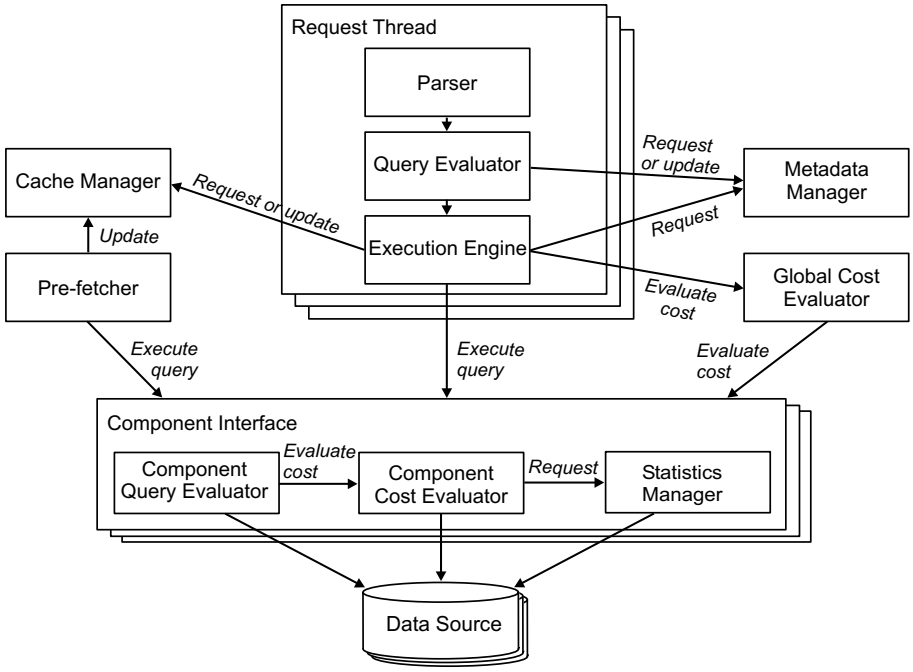


Figure 5: Architecture of the TSQL Evaluator.

and extension operators. 4) perform any additional grouping and/or selection on the combined result.

Generally, the performance of a TSQL query using external dimensions or measures will not be as good as if these data were integrated physically. However, for many queries, it is possible to achieve a performance close to that of the corresponding query on physically integrated data. We now discuss a number of optimization techniques. The three most significant ones are *caching* of full or partial query results, determining an *optimal evaluation strategy*, and *"inlining"* of external selection predicates.

The technique that is likely to provide the largest general performance improvement is *caching*. Basically, this is done by keeping some of the otherwise temporary tables produced during the evaluation of a TSQL query. In principle, all of the intermediate results produced during the evaluation could be kept for later use, but this is not a very efficient way of utilizing the cache space. Instead, we attempt to cache only the most useful intermediate results. However, which results are most useful depends e.g. on the exact pattern of usage for the system and thus, no single caching strategy is best for all situations. Instead, the caching strategy is based on assigning a *score* to each of the temporary tables and using this to determine which tables to expunge when the cache fills up. This score is determined from a number of factors that affect the quality of the cached table.

The evaluation strategy described above is not always optimal. Most significantly, the order in which decorations and extensions are performed may have a large impact on query performance. Generally, extension is performed before decoration which means that the dimension table is only joined with the combined fact table, rather than with both the OLAP result and the external measure table. Decorating with an external dimension may result in a much smaller fact table than before e.g. if the dimension is only used for selection. However, decoration may also increase the size if any dimension values are decorated by more than a single decoration value, i.e. if the hierarchy is non-strict. Extension does not change the number of rows in the result but it may change the total size of the table significantly. Which of these join orderings are the fastest is also affected by other aspects such as the size of the DBMS cache. To decide the order of decorations and extensions, the cost of each approach is estimated and the fastest is chosen. An estimation

may have to be performed for each combination of the external dimensions and measures, but since their number will usually be small (rarely more than 2–3), the overhead of the estimations will be limited.

The *external predicate inlining* technique [8] is used to evaluate selection predicates with references to external dimensions more efficiently. The straightforward way of evaluating these predicates is to retrieve only values for the lowest aggregation level from the cube, decorate with the external dimensions, evaluate the predicate, and finally calculate all the higher-level aggregates. If an external dimension is only used for selection it will have to be removed again before aggregation. To be able to decorate with an external dimension, the defining level of the dimension must be present in the intermediate cube result and this may increase its size significantly. Consequently, such a predicate may result in a considerable performance reduction. However, the inlining technique transforms the TSQL predicate such that it can be evaluated in the cube query without actually decorating with the external dimension. The basic idea is to evaluate the original predicate on the external dimension table, getting back a set of *literal data values*, and construct a new predicate that refers only to these literal values. For example, a predicate "[SupplierCities] IN ('Los Angeles','New York')" is translated to "[Supplier] IN ('A.A','B.B') which can be evaluated entirely in the OLAP component, resulting in much better performance.

When evaluating a TSQL query, it is first determined whether the whole or a part of the final result is in the cache. If a useful cached result is found, it is used. If no cached result can be used, the next step is to fetch the external data. Before starting new external queries for XML data, however, the cache is searched for external data that can be used in the evaluation, i.e. either external dimension and measure tables or raw XML data. For each subpredicate in the WHERE clause that refers to an external dimension, it is then determined whether the subpredicate should be inlined or evaluated after the OLAP query has completed. This is done by estimating and comparing the costs of the different strategies. It is then necessary to wait for the external dimension data for each of the subpredicates that should be inlined. While waiting, the inlining strategy is reconsidered if the estimated retrieval time for any of the external data to be inlined is exceeded significantly, or if any of the data that should not be inlined arrives before it was expected. When all external data to be inlined has been stored in tables, the OLAP query is constructed and started. Next, the same query is formulated in SQL and evaluated on the external measure tables when the measure data has been retrieved. Finally, the decoration/extension strategy is determined by comparing the costs of the different strategies. The decorations and extensions are then performed and the subpredicates that were not inlined are evaluated. Additional grouping may also be performed to remove levels only used for decoration.

The cost model consists of three submodels, one for the OLAP component, one for the XML components, and one for the temporary component. The OLAP cost model uses parameters such as query overhead time, disk read rate, fact and cube sizes, rollup fractions, predicate selectivity, and the use of pre-aggregation to determine OLAP cost estimates. The XML cost model uses parameters such as query overhead time, data read rates, network transfer rates, and documents parameters such as node sizes, fanouts, predicate selectivity, and path cardinality. The temporary component cost model is a standard relational cost model. The cost model parameters are generally estimated using *probing queries* and are *adapted* when the real queries are executed. Experiments show the cost models to work well.

We have performed a set of experiments with the prototype, using TPC-H data. The experiments showed that for many queries, our federated approach is a viable alternative to physical integration. Especially the inlining and caching techniques provided dramatic performance gains. Of course, some queries can never be evaluated efficiently in a federated setting.

7 Conclusion and Future Work

In this paper we have presented a flexible extension to the TARGIT OLAP client that allows dimensions and measures to be based on XML data residing outside the cube, e.g. on a Web-page. These dimensions

and measures can then, transparently to the end user, be used as ordinary dimensions and measures in the cube. The theoretical work presented in this paper has covered a flexible method for integrating *external measures* in OLAP databases as well as an formalization and extension of the TARGIT system's internal *multigranular* data model and query language with the ability to define and use external dimensions and measures. Additionally, a set of optimization techniques have been developed for the TARGIT system, which significantly extends the number of queries that can be evaluated efficiently. Most notably, a flexible caching approach has been proposed, and the inlining technique, which was also described in earlier work, has been adapted to the TARGIT platform. Another primary aspect of this work has been the construction of a working prototype, designed and implemented based on the theoretical results and the existing TARGIT system. We believe these contributions to be novel and interesting to both the database research and industry communities.

Future work will focus on evaluating and improving the prototype in order to include it in the TARGIT Analysis product, which is expected to happen over the next few releases. Another interesting direction could be to let *the system* choose between logical and physical integration of external data based on the usage pattern, the source's update frequency, the amount of processing required to perform the physical integration etc.

References

[1] S. Chawathe et al. The TSIMMIS Project: Integration of heterogeneous information sources. In *Proc. of IPSJ*, pp. 7–18, 1994.

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

[3] C. Graves et al. *Professional SQL Server 2000 Data Warehousing with Analysis Services*. Wrox Press, 2001.

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

[5] R. Kimball. *The Data Warehouse Toolkit*. Wiley, 1996.

[6] T. Lahiri, S. Abiteboul, and J. Widom. Ozone - Integrating Semistructured and Structured Data. In *Proc. of DBPL*, 1999.

[7] M. Middelfart. A Vision For Business Intelligence Systems in the Decade to Come. Technical report, Targit, 2001.

[8] D. Pedersen, K. Riis, and T. B. Pedersen. XML-Extended OLAP Querying. In *Proc. of SSDBM*, pp. 195–206, 2002.

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

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

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

[12] Targit A/S. `www.targit.com`. Current as of June 30th, 2003.

[13] W3C. XML Path Language (XPath) Version 1.0. www.w3.org/TR/xpath. Current as of June 30th, 2003.

[14] W3C. Extensible Markup Language (XML) 1.0 (Second Edition). www.w3.org/TR/REC-xml. Current as of June 30th, 2003.

[15] W3C. XQuery 1.0: An XML Query Language. www.w3.org/TR/xquery/. Current as of June 30th, 2003.