RAPPORT DE RECHERCHE

L R I

DATA ANALYSIS BASED ON FUNCTIONAL DEPENDENCIES

SPYRATOS N / SIMONENKO E / SUGIBUCHI T

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

10/2008

Rapport de Recherche N° 1501

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

Data Analysis Based on Functional Dependencies

Nicolas Spyratos
LRI-CNRS UMR 8623
Université Paris-Sud XI
91405 Orsay Cedex, France
spyratos@lri.fr

Ekaterina Simonenko
LRI-CNRS UMR 8623
Université Paris-Sud XI
91405 Orsay Cedex, France
simonenk@lri.fr

Tsuyoshi Sugibuchi
LRI-CNRS UMR 8623
Université Paris-Sud XI
91405 Orsay Cedex, France
buchi@lri.fr

ABSTRACT

We present a novel approach to the design of schemas appropriate for data analysis; we call such schemas "analysis contexts". Roughly speaking, an analysis context is a set of paths with common origin, in which the nodes are attributes and the edges are functional dependencies. Our main contributions are (a) an algorithm for generating all analysis contexts embodied in a set of attributes and a set of functional dependencies over the attributes, (b) a functional language in which analysis queries can be formulated within a context, (c) a formal approach to query optimization, and (d) an algorithm for mapping an analysis context and its associated functional language to a relational star schema for the efficient evaluation of analysis queries. Altogether, our main contribution is the proposal of a schema design method for analytic processing that parallels (and complements) the schema design process in transactional databases.

Keywords

Data Analysis, Functional Dependencies, Data Warehouse, OLAP Query, Star Schema, Schema Design, Schema Transformation

1. INTRODUCTION

In decision-support systems, in order to extract useful information from the data of an application, it is necessary to analyse large amounts of detailed transactional data, accumulated over time - typically over a period of several months. The data is usually stored in a so-called "data warehouse", and it is analysed along various dimensions and at various levels in each dimension [6].

A data warehouse functions just like a usual database, with the following important differences: (a) the data is not production data but the result of integration of production data coming from various sources, (b) the data is historic that is data accumulated over time, (c) access to the data warehouse by analysts is almost exclusively for reading and *not* for writing and (d) changes of data happen only at the sources, and such changes are propagated periodically to the data warehouse.

The end users of a data warehouse are mainly analysts and decision makers, who almost invariably ask for data aggregations such as "total sales by store", or "average sales by city and product category". On-Line Analytic Processing, or OLAP for short, is the main activity carried out by analysts and decision makers [1, 7]. In fact, the basic requirements of data analysts are (a) a data schema that is easy to understand and (b) a query language in which to express easily complex data analysis tasks. The so called "dimensional schemas" (such as snow-flake schemas, star schemas etc.) and the various SQL extensions (such as "grouping sets", "rollup", "cube" etc.) were introduced precisely to satisfy these requirements [6].

In this paper, we present a novel approach to data analysis based on functional dependencies. Given a set U of attributes and a set F of functional dependencies, our approach relies on some basic notions from relational database theory to define the notion of *analysis context* (or *context*, for short). Roughly speaking, a context is a set of paths with common origin, and data analysis in our model always takes place in a context.

In this section we illustrate our approach through examples. As a first example, consider a database which accumulates documents over time (eg. a digital library). Each document is identified by its URI and described by two attributes:
$Topic$, whose values are keywords describing document content (eg. drama, poetry, etc.)
$Hits$, whose values are integers representing number of accesses to the document.
Thus each document URI in the library is associated with one keyword (its topic description) and one integer (its number of hits). Therefore we have two functional dependencies
$t : URI \rightarrow Topic$ and $h : URI \rightarrow Hits$
(we need the labels t and h for later reference).

We can describe this application by a graph, as shown in Figure 1.(a), where the attributes URI, $Topic$ and $Hits$ are the nodes and the functional dependencies t and h are the edges. This graph is a first, rudimentary example of what we call analysis context. Its *origin* is the attribute URI which also happens to be a key (in this example). Roughly speaking, the origin of a context represents the objects of interest, while all other nodes describe attributes of the objects.
In our approach, we interpret the edges $t : URI \rightarrow Topic$ and $h : URI \rightarrow Hits$ of the context as function signatures,

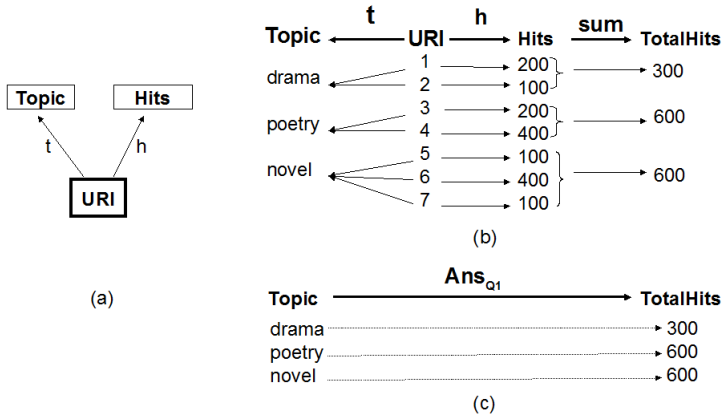

Figure 1: Example of an application represented by a graph, and a query evaluation.

and their extensions as the (current) instance of the context. Figure 1.(b) shows an example of (current) instance. This instance consists of the two functions t and h, and represents all documents accumulated so far in the library. It contains 9 document URIs, each associated with its topic and its number of hits through the functions t and h, resp. (for simplicity, we represent URIs as integers); for example, document 3 is associated with "Poetry" as topic and with 200 as number of hits (i.e. t(3)= Poetry and h(3)= 200).

Suppose now that we want to analyse document usage, say by finding the total number of hits by topic, that is by evaluating the answer to the following query:

Q_1: "total number of hits by topic."

In our approach, in order to answer this query we proceed in three steps as follows:
Grouping: we invert the function t, thus grouping the URIs by topic;
Measuring: in each group, we apply the function h to each URI of the group to find the corresponding number of hits;
Aggregation: in each group, we sum up the results of measuring to have the total number of hits for that group (as shown in Figure 1.(b)).
The final result is shown in Figure 1.(c), and it is a function from $Topic$ to a new attribute that we call $TotalHits$. This function associates each topic with the total number of hits for that topic. In other words the answer to Q_1 is the following function:

$$Ans_{Q_1} : Topic \rightarrow TotalHits$$

such that $Ans_{Q_1}(x)$ is the total number of hits, for each topic x. For example,

$$Ans_{Q_1}(Drama) = 300 \text{ and } Ans_{Q_1}(Poetry) = 600.$$

This pattern of grouping a set of objects by inverting a function defined on them; then measuring a property in each group by applying a second function also defined on the objects; and finally aggregating the measures in each group by applying an operation on the measures, constitutes the basic pattern of our approach.

It should be clear from the previous example that the specification of query Q_1 requires three parameters, a function such as t for classifying the URIs by topic, a function such as h for measuring the number of hits by URI, and an operation such as "sum" for aggregating the measured numbers of hits. Therefore Q_1 can be specified as a triple:

$$Q_1 =< t, h, sum >.$$

Notice however that t and h have the origin of the context as their *common* domain of definition, and that this condition is indispensable in order to compute the answer. Moreover, notice that the operation "sum" is an operation which is possible to apply over the *range* of h (i.e. over the integers), and that this condition is also indispensable in order to compute the answer.

In view of the previous discussion, in our approach, we define an analytic query over a context to be a triple $Q =< c, m, op >$, where c and m are edges of the context having the origin as their common domain of definition, and op is an operation which is possible to apply over the range of m. We refer to the function c as the *classifier* or the *grouping function* of Q and to the function m as the *measure*.

It is interesting to note that, following the above definition, one can interchange the roles of c and m to obtain a different analytic query (provided of course that the operation in the resulting query is possible to apply over the range of the new measure). For example, one can interchange the roles of t and h in query Q_1 to obtain the query $Q'_1 =< h, t, sum >$. However, this query is not well formed since the operation

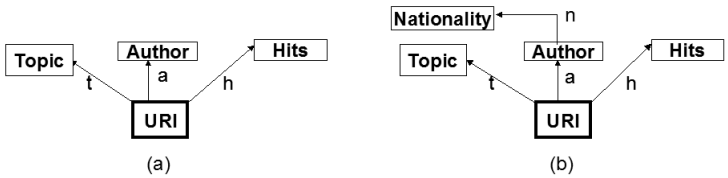

Figure 2: Adding Author and Nationality.

"*sum*" cannot be applied over the range of t (i.e. over $Topic$, as topics can't be summed). If one changes the operation from "*sum*" to "*count*" then one obtains a well formed query, namely

$$Q_1'' =< h, t, count >$$

asking for the number of topics by number of hits.
Continuing with our example, suppose now that, in addition to topic description and number of hits, each document is also associated with an author. Then we obtain a new context as shown in Figure 2.(a), where we added the edge $a : URI \rightarrow Author$. We can now formulate the following query, asking for the total number of hits per author:

$$Q_2 =< a, h, sum >.$$

The answer will be a function from authors to integers:

$$Ans_{Q_2} : Author \rightarrow TotHits$$

such that $Ans_{Q_2(x)}$ is the total number of hits, for each author x.
Now, however, it makes sense to also ask the following query:

Q_3: "total number of hits by topic-author pair".

This time the grouping of URIs will be done according to a function derived from t and a, which associates each URI with a topic-author pair. This function is called the *pairing* of t and a, it is denoted by $t \wedge a$, and it is defined as follows: $t \wedge a : URI \rightarrow Topic \times Author$ such that $(t \wedge a)(x) =< t(x), a(x) >$.
During evaluation of Q_3, the pairing $t \wedge a$ will group together all URIs having the same topic and the same author, and the answer will associate each topic-author pair with a total number of hits. In other words, compared to Q_1, the only change is that the function t of Q_1 is now replaced by the function $t \wedge a$; other than that, the evaluation of Q_3 proceeds in exactly the same way as for Q_1. Therefore, Q_3 is specified as follows:

$$Q_3 =< t \wedge a, h, sum >.$$

And its answer will be a function from topic-author pairs to integers:

$$Ans_{Q_3} : Topic \times Author \rightarrow TotHits$$

such that $Ans_{Q_3}((x, y))$ is the total number of hits, for each topic-author pair (x, y).

As another example, suppose that we also add the nationality of each author. Then we obtain a new context as shown in Figure 2.(b). We can now ask the following analytic query:

Q_4: "total number of hits by author's nationality".

This time, during evaluation, the grouping of URIs will be done according to a function derived from a and n using functional composition. The composition of a and n, denoted $n \circ a$, associates each URI with the corresponding author's nationality. During evaluation of Q_4, the function $n \circ a$ will group together all URIs having the same author nationality, and the answer will associate each author nationality with a total number of hits. In other words, compared to query Q_1, the only change is that the function t of Q_1 is now replaced by the function $n \circ a$; other than that, the evaluation of Q_4 proceeds in exactly the same way as for Q_1. Therefore Q_4 is specified as follows:

$$Q_4 =< n \circ a, h, sum >.$$

And its answer will be a function from $Nationality$ to $TotHits$:

$$Ans_{Q_4} : Nationality \rightarrow TotHits$$

such that $Ans_{Q_4}(x)$ is the total number of hits, for each nationality x.

As a final example, we can ask the following analytic query over the context of Figure 2.(b):

Q_5: "total number of hits by topic-nationality pair".

Again, all we have to do in order to evaluate this query is to replace the function t of Q_1 by the function $c \wedge (n \circ a)$. Therefore Q_5 is specified as follows:

$$Q_5 =< c \wedge (n \circ a), h, sum >.$$

And its answer will be a function from topic-nationality pairs to integers:

$$Ans_{Q_5} : TopicNationality \rightarrow TotHits$$

such that $Ans_{Q_5}((x, y))$ is the total number of hits, for each topic-nationality pair (x, y).

Notice that, in all the above examples of queries (Q_1 to Q_5), we can also restrict any of the functions involved to some desirable subset of its domain of definition, to form new analytic queries. In fact, the set of all operations on functions that we shall use to derive new functions from old constitutes what we shall call the functional algebra of a context. These operations are quite elementary: composition, pairing, restriction and projection. Yet, as we shall see, they allow us to associate each context with a powerful language of analytic queries.

In our examples so far we dealt with only one context. Before ending this introductory section, let's see an example in which we have several contexts sharing nodes. First, let us give a more precise definition of context. A context is the set of all paths starting at a specified node (this node being called the *origin* of the context).

Consider now the application described in Figure 3.(a) in the form of a graph of functional dependencies, in which we have students registered in schools (each student being registered in one and only one school), and donators making one or more donations to one or more schools (the remaining attributes are self-explanatory).
In this application, we have several contexts with nodes in common. For example, the $StudentId$ context (i.e. the context with $StudentID$ as its origin) and the $DonationId$ context have $School$, $City$ and $State$ as common nodes. These two contexts are shown in Figures 3.(b) and 3.(c). The $StudentId$ context would be selected if one wanted to analyze data pertaining to students, while the $DonationId$ context would be selected if one wanted to analyze data pertaining to donations. Another example of context is the $School$ context; this context would be selected if one wanted to analyze data pertaining to schools.
In general, it is up to the user to select a context suitable for a specific application.

As a final remark, we mention that the origin of a context is not necessarily a key.
Indeed, in our examples of Figure 3, neither of the attributes $StudentId$ or $DonationId$ is a key. The only key in this application is the pair $< StudentId, DonationId >$, which can not be used as origin for a context, as it does not appear explicitly in the graph of Figure 3.

The rest of the paper is organized as follows. In section 2 we give the formal definition of context and present an algorithm for generating all contexts embodied in a set of attributes and a set of functional dependencies over the attributes. In section 3 we show how a context can be associated with a functional language in which analytic queries can be formulated. In section 4, we present an approach to query optimization based on the lattice of partitions of the context's origin. In section 5 we present an algorithm for mapping a context and its associated functional language to a relational star schema for the efficient evaluation of analytic queries. In section 6 we discuss implementation issues and describe a prototype (currently under development) which allows users to (a) formulate analytic queries by "clicking" on the graphical presentation of a context, and (b) visualize the answer in various forms and further exploit it. Finally, in section 7, we offer some concluding remarks and discuss perspectives of this work.

2. THE FORMAL DEFINITION OF CONTEXT

The formal definition of context uses a few well known concepts from the relational model that we recall next (see [6] for more details).
Let U be a set of attributes and F a set of functional dependencies over U. Without loss of generality, in all our definitions, we assume that the set F is *reduced*. This means that F has the following properties: (a) it contains no trivial dependencies, (b) it is non redundant, that is no dependency in F can be derived from other dependencies in F and (c) for each dependency $X \rightarrow Y$ in F, Y is a single attribute and there is no strict subset X' of X such that the dependency $X' \rightarrow Y$ can be derived from other dependencies in F. It follows that among the attribute sets appearing in F, only those that appear on the left hand side of a dependency can have more than one attribute. Finally, we note that any given set G of functional dependencies can always be reduced to an equivalent set F having the above properties. Such a set F is called a *reduction* of G, and there might be two or more different reductions for a given G.

Based on the attribute sets and dependencies of F, we define now a graph, that we shall call the *dependency graph*, denoted $Graph(F)$:

1. *Nodes:*
   (a) Each attribute set appearing in F is a node of $Graph(F)$.
   (b) For each attribute set X appearing in F with more than one attribute, say $X = \{A_1, A_2, ..., A_k\}$, the following are nodes of $Graph(F)$: $A_1, A_2, ..., A_k$.
2. *Edges:*
   (a) Each functional dependency $X \rightarrow Y$ in F is an edge of $Graph(F)$.
   (b) For each attribute set X appearing in F with more than one attribute, say $X = \{A_1, A_2, ..., A_k\}$, the following are edges of $Graph(F)$: $X \rightarrow A_1, X \rightarrow A_2, ..., X \rightarrow A_k$.

As an example consider the following pair (U, F):

$U = \{A, B, C, D, F, E, G, H, K\}$,
$F = \{AB \rightarrow C, A \rightarrow D, B \rightarrow E, BC \rightarrow A, D \rightarrow A, C \rightarrow F, F \rightarrow G, A \rightarrow E, D \rightarrow E, HK \rightarrow D, HK \rightarrow F\}$

Figure 4.(a) shows the corresponding dependency graph, $Graph(F)$. We note that the graph $Graph(F)$ can be a cyclic and/or disconnected graph.
Our definition of context relies on what we call the "quotient graph" of F, a graph defined based on $Graph(F)$ and using the notion of "closure" of an attribute set.
We recall from relational database theory [6] that (a) the closure of an attribute set X, with respect to a set F of functional dependencies, denoted $Cl(X, F)$, is the set of all attributes that one can derive using Armstrong's inference rules for functional dependencies, and (b) that there is a linear time algorithm for computing closures, known as the *closure algorithm* (linear in the size of F, where the size of each functional dependency is the number of attributes involved [6]).
Using closures, we can define a binary relation "$\leq$" over attribute sets as follows: for all attribute sets X, Y define: $X \leq Y$ if and only if $Cl(X, F) \supseteq Cl(Y, F)$. Clearly, the relation "$\leq$" is reflexive and transitive but not anti-symmetric (hence "$\leq$" is a partial pre-ordering over attribute sets). Now, using "$\leq$" we can define an equivalence relation over attribute sets as follows: for all attribute sets X, Y define: $X \equiv Y$ if and only if $Cl(X, F) = Cl(X, F)$. Indeed, the relation "$\equiv$" is reflexive, symmetric and transitive, hence an

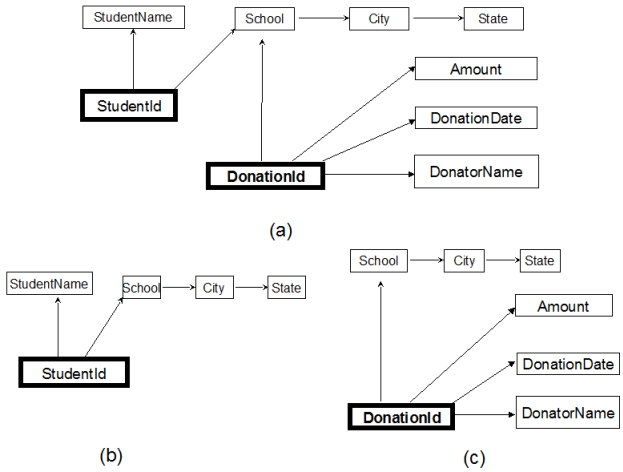

Figure 3: An example illustrating a choice of analysis context.

equivalence relation. Using functional dependencies, equivalence can be also characterized as follows: $X \equiv Y$ if both dependencies, $X \rightarrow Y$ and $Y \rightarrow X$ are implied by F using Armstrong's inference rules [6].
In what follows, given an attribute set X, we shall denote its equivalence class by $\mathbf{X}_F$, or simply by $\mathbf{X}$ if F is understood.

Equivalence between attribute sets is an interesting concept, in the sense that equivalent attribute sets are just different descriptions of some underlying objects. For example, consider the attributes $Product(P)$, $PriceInDollars(PD)$ and $PriceInYen(PY)$ along with the following dependencies: $P \rightarrow PD$, $P \rightarrow PY$, $PD \rightarrow PY$, $PY \rightarrow PD$. Then the attributes PD and PY are equivalent, and this means that we can use the one or the other interchangeably, depending on the needs of the specific application (Dollars in the US and Yen in Japan).

We are now ready to define the basic notion of quotient graph of F, on which is based the formal definition of context. Actually the nodes of the quotient graph are the equivalence classes of the nodes of $Graph(F)$, and the edges of the quotient graph are the pre-orderings between the equivalence classes.
Definition 1 - Quotient Graph
Let U be a set of attributes and F a (reduced) set of functional dependencies over U. The quotient graph of F, denoted $\mathbf{F}$, is a graph defined as follows:

- *Nodes:* For each node X of $Graph(F)$, $\mathbf{X}$ is a node of $\mathbf{F}$.

- *Edges:* For any pair of nodes $\mathbf{X}$ and $\mathbf{Y}$ in $\mathbf{F}$, there is an edge from $\mathbf{X}$ to $\mathbf{Y}$ if and only if $Cl(X, F) \supseteq Cl(Y, F)$.

In the previous example, we have the following equivalences:

$$A \equiv D \text{ and } AB \equiv BC$$

This is easy to verify, by computing closures:

$$AB^+ = \mathbf{U},\ A^+ = ADE,\ B^+ = BE,\ BC^+ = \mathbf{U},$$
$$D^+ = ADE,\ C^+ = CF,\ H^+ = ADFGEHK.$$

Figure 4.(b) shows the corresponding quotient graph.
It is not difficult to see that the construction of the quotient graph requires n computations of closure, where n is the number of attribute sets appearing in $Graph(F)$. Indeed, once the n closures have been computed, all nodes of $Graph(F)$, having the same closure, form one node of $\mathbf{F}$. Moreover, there is an edge from node $\mathbf{X}$ to node $\mathbf{Y}$ in $\mathbf{F}$, if and only if the (common) closure of the attribute sets in $\mathbf{X}$ is a superset of the (common) closure of the attribute sets in $\mathbf{Y}$; this follows from the well known property of the relational model: F implies the dependency $X \rightarrow Y$ if and only if the closure of X is a superset of the closure of Y.
Therefore the complexity of computing the quotient graph (measured in number of closure computations) is linear to the number of attribute sets appearing in $Graph(F)$.

We note that, although the graph F might be cyclic, the quotient graph of F has no cycles (i.e. it is acyclic). Roughly speaking, this is due to the fact that all cycles that appear in $Graph(F)$ are "hidden" within the nodes of the quotient graph. We also note that, as the quotient graph is acyclic, it has at least one root.

Our formal definition of context is based on the notion of "context class" that we define now.

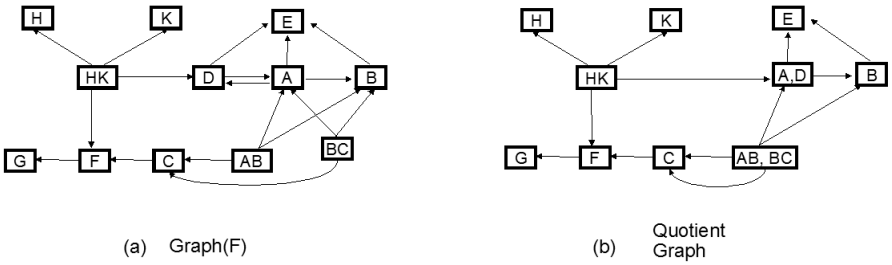

Figure 4: Example of Graph(F) and corresponding Quotient Graph.

Definition 2 - Context Class
Let U be a set of attributes, F a (reduced) set of functional dependencies over U, and **F** the quotient graph of F. A *context class* of F is the set of all paths starting at a given node **O** of **F**. The node **O** is called the *origin* of the context class.
For example, in the quotient graph of Figure 4.(b), consider the node $\mathbf{O} = \{AB, BC\}$. The context class with origin O is shown in Figure 5.(a)

Our definition of context relies on that of context class. Roughly speaking, given a context class, a context is defined by selecting an attribute set in each of the nodes of the context class and keeping the edges of the context class.

Definition 3 - Context
Let U be a set of attributes, F a (reduced) set of functional dependencies over U, **F** the quotient graph of F. Let **Cxt** be a context class in **F** with origin **O**.
A *context* in **Cxt** is an acyclic graph Cxt defined as follows:

- *Nodes* of Cxt: In each node **N** of **Cxt**, select an attribute set $N \in \mathbf{N}$; the attribute set O selected from **O** is called the origin of Cxt.
- *Edges* of Cxt: For any pair of nodes X and Y of Cxt, there is an edge $X \rightarrow Y$, if and only if there is an edge $\mathbf{X} \rightarrow \mathbf{Y}$ in **F**.

Figure 5.(b) shows a context defined from the context class of Figure 5.(a), by selecting the attribute set BC from the origin of the context class, and the attribute A from the node $\{A, D\}$ of the context class (note that the origin and the node $\{A, D\}$ are the only nodes of the context class where a choice is possible).

We note that, in a context, each node X other than the origin O is a single attribute (because F is reduced), and as such it is associated to a set of values, or domain, denoted $dom(X)$. However, the origin O may contain two or more attributes, say $O = \{A_1, A_2, ..., A_n\}$; nevertheless, we shall consider that O is also associated with a domain, namely $dom(O) = dom(A_1) \times dom(A_2) \times ... \times dom(A_n)$.
In the following section, we define a query language for performing data analysis once a context is selected. How such a selection is made is a topic that lies outside the scope of the present paper.

3. THE QUERY LANGUAGE OF A CONTEXT

As we mentioned in the introduction, we view a context as a set of function signatures and a context instance as a set of (finite) extensions, one for each function signature. In this section, we keep with this view and we define a language in which analytic queries can be formulated over the context and evaluated over the context instance. We shall illustrate the concepts introduced by the following example that we shall use as our running example for the rest of the paper.

Running example
A big catering company delivers various products to retail stores over the whole country. The following data appears on the delivery invoice:

- the invoice number
- the date of delivery
- the store identifier

and for each type of product:

- the product reference (e.g. "Coca Light")
- and the number of units delivered of that product (e.g. Coca Light: 1600 cans)

These data are recorded in the database of the catering company, and accumulated over long periods of time with the purpose of analyzing them in order to improve the company's delivery service. More specifically, the analyses performed are:

- by date, by month and by year

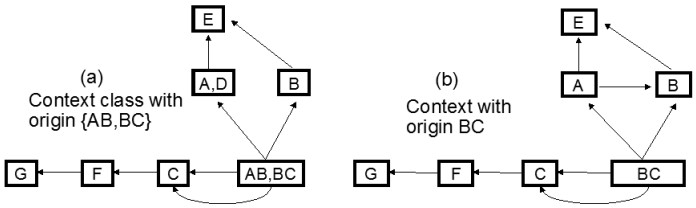

Figure 5: A Context class and a Context.

- by store, by city and by region
- by supplier and by product category
- by combinations thereof, such as by date and store, or by month and region etc.

We assume that there is at most one delivery per day, so the following dependency holds: $Date, Store, Prod \rightarrow Number$. Therefore the triple $\{Date, store, Prod\}$ is a key. We also assume that a supplier might supply products in two or more categories and that a product category might be supplied by two or more suppliers.
The context concerning this application is shown in Figure 6, where O stands for the triple $\{Date, Store, Prod\}$. We shall use this context in order to introduce the basic concepts of the query language.

In our explanations we shall use the notation $f : X \rightarrow Y$ to denote an edge with *label* f, *source* X and *target* Y; similarly, we shall talk of the source and the target of a path.

3.1 Functional database

Given a context S, a *database* over S is a function δ that associates :

- each node N of S with a finite subset $\delta(N)$ of $dom(N)$, and
- each arrow $f : X \rightarrow Y$ of S with a total function $\delta(f) : \delta(X) \rightarrow \delta(Y)$.

In order to simplify notation, we shall omit the symbol δ and we shall use the expression "function $f : X \rightarrow Y$" to mean "function $\delta(f) : \delta(X) \rightarrow \delta(Y)$".
We note that the fact that all functions in the database are total imposes the following constraint:

referential constraint : for every pair of functions of the form $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ we must have $range(f) \subseteq def(g)$.

Roughly speaking, the schema is seen as a set of function signatures and the database stores their extensions.

Several remarks are in order here. First, in order to simplify the presentation, we adopt the following abuse of notation: we use an arrow label such as f to denote both the arrow f and the function $\delta(f)$ assigned to f by δ. Similarly, we use an attribute label such as X to denote both the attribute X and the finite set $\delta(X)$ assigned to X by δ. This should create no confusion, as more often than not the context will resolve ambiguity.
Second, the definition of a database requires that all functions assigned by the database δ to the arrows of S be total functions. This restriction could be relaxed, by endowing each attribute domain with a bottom element $\perp$ (meaning "undefined") and requiring that for any function $f : X \rightarrow Y$ we have (a) $f(\perp) = \perp$, that is "bottom can only map to bottom", and (b) if $x \notin def(f)$ then $f(x) = \perp$. Under these assumptions, the functions can again be considered as total functions. However, the resulting theory would be more involved and would certainly obscure some of the important points that we would like to bring forward concerning analytic queries.

3.2 The Functional Algebra

In order to combine the function extensions in the database of a context, we need a set of operations on functions that we call the *functional algebra*. This algebra is similar to the one considered in the functional model of databases [3] and comprises four operations. Each operation takes as input one or two functions and returns a new function as a result:

Composition : takes as input two functions, f and g, such that $range(f) \subseteq def(g)$, and returns a function $g \circ f : def(f) \rightarrow range(g)$ defined by

$$g \circ f(x) = g(f(x)), \text{ for all } x \in def(f).$$

Pairing : takes as input two functions, f and g, such that $def(f) = def(g)$, and returns a function $f \wedge g : def(f) \rightarrow range(f) \times range(g)$ defined by

$$f \wedge g(x) = \langle f(x), g(x) \rangle, \text{ for all } x \in def(f).$$

Projection : it's the usual operation on the cartesian product of sets.

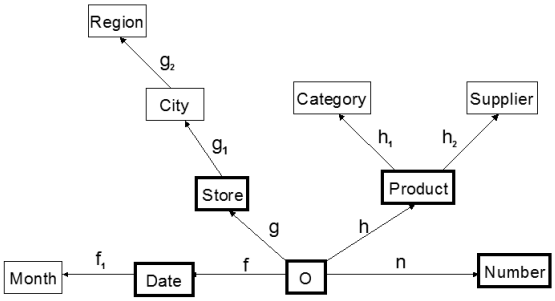

Figure 6: The context S of the running example.

Restriction : takes as input a function $f : X \to Y$ and a set $E \subseteq def(f)$, and returns a function $f/_E : E \to Y$ defined by

$$f/_E(x) = f(x), \text{ for all } x \in E.$$

The following proposition states an important property of the functional algebra.

Proposition 1
For every pair of functions $f : X \to Y$ and $g : X \to Z$, we have

$$f = \pi_Y \circ (f \wedge g) \text{ and } g = \pi_Z \circ (f \wedge g).$$

Nota : $\pi_Y()$ and $\pi_Z()$ are the projection functions over $Y \times Z$, defined by: $\pi_Y(y, z) = y$ and $\pi_Z(y, z) = z$, for all pairs $(y, z) \in Y \times Z$.

Path expressions
Given a context S, a *path expression* over S is a well formed expression whose operands are arrows from S and whose operations are those of the functional algebra. Every path expression e is associated with a source and a target, defined recursively, based on the notions of source and target of the arrows in S. For instance, in our running example, if $e_1 = g_2 \circ g_1$ then $source(e_1) = Store$ and $target(e_1) = Region$; similarly, if $e_2 = (g_1 \circ g) \wedge f$ then $source(e_2) = O$ and $target(e_2) = City \times Date$ (for a formal definition of a path expression see [8]).

Given a path expression e and a database δ over S, the evaluation of e with respect to δ is done in two steps, as follows:

1. replace each arrow f appearing in e by the function $\delta(f)$ that the database δ associates with f
2. perform the operations of the functional algebra as indicated in e.

Note that the result of the evaluation is always a function; therefore we have a closure property as in the case of the relational algebra.

A particular kind of path expression will be of interest, namely the one that corresponds to the empty projection function. To understand the nature of this projection, recall that, given a Cartesian product of k sets, say $A_1 \times ... \times A_k$, there are as many projection functions as there are subsets of the set $A_1, ..., A_k$. The projection function that corresponds to the empty set is the one that we call the empty projection function, hence we denote it by $\pi_\emptyset$. Clearly, following the definition of a projection function, the function $\pi_\emptyset$ is a constant function as it associates every tuple of $A_1 \times ... \times A_k$ with the empty tuple; we shall denote the empty tuple by λ. Therefore, $\pi_\emptyset(t) = \{\lambda\}$, for all $t \in A_1 \times ... \times A_k$ (assume $A_1 \times ... \times A_k$ is non-empty). In view of our previous discussion, we introduce a particular path expression, the *empty path expression*, which will always be associated with the empty projection function, in any database δ. We denote the empty path expression by ε_X, where X denotes the source of the empty path expression (its target being always interpreted as $\{\lambda\}$). The empty path expression with source O will be simply denoted by ε.

3.3 OLAP query

Given a context S, an *OLAP query* over S is a triple

$$Q =< c, m, op >, \text{ where}$$

- c and m are path expressions over S such that $source(c) = source(m)$ and
- op is an operation among those authorized over the target of m.

For example, in the schema S of our running example (Figure 6), the following is an OLAP query :

$$Q = (g \wedge (h_2 \circ h), n, sum)$$

In this query we have :
- $c = g \wedge (h_2 \circ h)$, $m = n$ and $op = sum$
- $source(g \wedge (h_2 \circ h)) = source(n) = O$,
sum is an authorized operation over the target of n (which is N, with $dom(N) = Int$).

Given an OLAP query $Q =< c, m, op >$, we call c the *classifier*, m the *measure* and op the *operation* (or the *aggregator*) of Q. Moreover, we call the target of c the *classification level* (or the *grouping level*), and the target of m the *measurement level*. Note that the classification level, or the measurement level, might be composed of other simpler levels. This is the case in our previous example where $target(c) = Store \times Supplier$; the classification level is $Store \times Supplier$ and it is composed of $Store$ and $Supplier$.

3.4 Evaluation of an OLAP query

Given an OLAP query $Q =< c, m, op >$ and a database δ over S, the answer to Q with respect to δ is a function $ans_{Q,\delta} : range(c) \rightarrow target(op)$ computed in two steps, as follows:

1. Evaluate the path expressions c and m with respect to δ.
   {This step returns two functions that we also denote as c and m. Let's call X the common domain of definition of the functions c and m, that is $X = source(c) = source(m)$; and let $\{y_1, ..., y_n\}$ be the set of values of c, that is $Y = range(c) = \{y_1, ..., y_n\}$}
2. For each value $y \in range(c)$ do
   begin
   (a) ***Grouping***
   compute the inverse $c^{-1}(y)$
   {let $c^{-1}(y) = \{x_1, ..., x_r\}$}
   (b) ***Measurement***
   for each $x \in c^{-1}(y)$ do compute $m(x)$
   {this step returns a tuple
   $t(y) =< m(x_1), ..., m(x_r) >$}
   (c) ***Aggregation***
   apply the operation op to the tuple $t(y)$
   {call the result $Res(y)$, that is $Res(y) = op(t(y))$}
   (d) ***Answer***
   define $ans_{Q,\delta}$, $(y) = Res(y)$
   end

We note that the variable Res used in the evaluation algorithm (step 2.(c)) does not appear as a node of the context; it is actually an auxiliary variable defined by the user in order to receive the results of the computation. Moreover, as δ is understood (it's always the current database), we shall drop δ from the notation of the answer, and we shall use the symbol ans_Q or $ans(Q)$.

A particular form of OLAP query is the following, where ε denotes the empty expression :

$$Q =< \varepsilon, m, op >$$

During its evaluation, step 1 of the evaluation algorithm returns a constant function with λ as its only value; step 2.(a) (grouping) returns just one group $c^{-1}(\lambda) = \{X\}$; step 2.(b) (measurement) returns the images under m of all elements of X; step 2.(c) (aggregation) applies the operation on all images to obtain a single result $Res(\lambda)$; and step 2.(d) (answer) associates every element of X to $Res(\lambda)$. Therefore, the answer $ans_{Q,\delta}$, is itself a constant function.
As an example, the query

$$Q =< \varepsilon, n, Sum >$$

will return the total number of items delivered (i.e. the total number of items present in the database).

When the target of the classifier c is a Cartesian product, say $A_1 \times, ..., \times A_k$, then the representation of the answer $ans_{Q,\delta} : range(c) \rightarrow target(op)$ by cross tabulation is usually called a "data cube" [1, 7].

A final but important remark is that our approach treats the classifier and the measure of an OLAP query in a symmetric manner. Indeed, as the classifier and the measure of an OLAP query $Q =< c, m, op >$ *both* are path expressions, if we interchange them we obtain a different but valid OLAP query, possibly after changing the operation op. For example, consider the query $Q =< g, n, sum >$, which asks for the total number of items delivered by store. If we interchange g and n, and use "count" instead of "sum", then we obtain the query $Q =< n, g, count >$, which asks for the number of stores by number of units delivered (i.e. given a number of items, how many stores had this number of items delivered to them). Note that, in the query Q, we used "*count*" as an operation, since this is the only operation allowed on the target of the measure g (which is a set of store references).

3.5 Optimization Issues

As we have seen in the previous section, the partition of O resulting from the grouping step, plays a crucial role in determining the answer. Given a query $Q = (u, v, op)$, the partition induced by the function u on the set of objects O is called the *support* of Q and it is denoted as s_Q. Query optimization consists in using the answer of an already evaluated query in order to evaluate the answer of a new query *without* passing over the data again. In our model we use the lattice of partitions of the set O as the formal tool to achieve such optimization.

Definition 8 - The Lattice of Partitions
Let p, p' be two partitions of O. We say that p is *finer* than p', denoted $p \leq p'$, if for each group G in p there is a group G' in p' such that $G \subseteq G'$.

One can show that $\leq$ is a partial order over the set of all partitions of O (i.e. a reflexive, transitive and anti-symmetric binary relation over partitions). Under this ordering, the set of all partitions of O becomes a lattice in which the partition $\{O\}$ is the *top* (the coarsest partition) and the partition $\{\{o\}/o \in O\}$ is the *bottom* (the finest partition).

To see how this lattice can be used to achieve optimization, consider a query $Q = (u, v, op)$ which has already been evaluated. As we have explained earlier, if $y_1, .., y_k$ are the

values in the range of u, then the support of Q is the following partition of O:

$$s_Q = \{u^{-1}(y_i)/i = 1, .., k\}$$

Based on the support, the answer to Q is expressed as follows:

$$ans_Q(y_i) = op(v(u^{-1}(y_i))), i = 1, .., k.$$

Now, suppose that a new query Q'= (u', v', op') comes in and we want to evaluate its answer. We claim that if $s_Q \leq s_{Q'}$ then the answer to Q' can be expressed in terms of the support of Q. This is based on a simple fact, which follows immediately from the definition of the partition ordering:

Fact : if $s_Q \leq s_{Q'}$ then each group G' in $s_{Q'}$ is the union of groups from s_Q.
As a result, if G'= $G_1 \cup \ldots \cup G_j$ then
$op'(v'(G')) = op'(v'(G_1 \cup \ldots \cup G_j)) = op'$ $(v'$ (G_1) , ..., $(v'$ $(G_j))$.

As the support of Q has already been computed (and is available), we can apply v' and then op' "off-line" (i.e. *without* passing over the data again). Moreover, if v= v' then we can reuse the measurements of Q as well.
That is, if v= v' then we have:

$$op'\ (v'\ (G_1), \ldots, (v'\ (G_j))= op'\ (v\ (G_1)\ , \ldots, (v\ (G_j))$$

Finally, if in addition op= op' then we can reuse even the summarizations of Q, provided that the following property holds:

$$op(v(G_1), \ldots, v(G_j))= op(op(v(G_1)), \ldots, op(v(G_j))$$

One can show that this property holds for most of the usual operations, namely "sum", "count", "max", and "min", but not for "avg". For example,

$$sum(2, 4, 6, 7) = sum(sum(2, 4),\ (sum(6, 7)),$$
while $avg(2, 4, 6, 7) \neq avg(avg(2, 4),\ avg(6, 7))$.

However, all the above results hold only if we know that $s_Q \leq s'_Q$ (see Fact above), so the question is: given two queries, Q and Q', can we decide whether $s_Q \leq s'_Q$?
To answer this question, we observe first that the classifier u of an OLAP query is essentially the pairing of a number of compositions along path expressions. Therefore it is sufficient to answer the above question for two separate cases: when the classifier is a composition and when the classifier is a pairing. The following proposition provides the answers.

Proposition 2 - Comparing Classifiers

- Grouping by Composition
  Let $Q = (u, v, op)$ and $Q' = (u', v', op')$ be two OLAP queries such $u = p$ and $u' = q' \circ p$, where p and q' are path expressions. Then $s_Q \leq s_{Q'}$.

- Grouping by Pairing
  Let $Q = (u, v, op)$ and $Q' = (u', v', op')$ be two OLAP queries such $u = p \wedge q$ and $u' = p$, where p and q are path expressions. Then $s_Q \leq s_{Q'}$.

In our running example, if $Q = (g, q, sum)$ and Q'= $(g_1 \circ g, q, sum)$, then $s_Q \leq s_{Q'}$, therefore the answer of Q' can be computed from that of Q. Similarly, if $Q = (g \wedge h, q, sum)$ and Q'= (g, q, sum), then again $s_Q \leq s_{Q'}$, and the answer of Q' can be computed from that of Q.
The proof of the above proposition follows from properties of function inverses, as stated in the following proposition.

Proposition 3 - Properties of Inverses
composition:
Let $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ be two functions. Then for all $z \in range(g \circ f)$ we have:

$$(g \circ f)^{-1}(z) = \cup\{f^{-1}(y)/y \in g^{-1}(z))$$

that is, a z-group under $g \circ f$ is the union of all y-groups under f, where y ranges over the z-group under g

pairing:
Let $f : X \rightarrow Y$ and $g : X \rightarrow Z$ be two functions. Then for all $(y, z) \in range(f \wedge g)$ we have:

$$(f \wedge g)^{-1}((y, z)) = f^{-1}(y) \cap g^{-1}(z)$$

Lack of space does not allow further details on optimization.

4. MAPPING TO THE RELATIONAL MODEL

The context model that we presented in the previous section can certainly be used as is to model an application in order to perform data analysis. Moreover, a context database can be implemented using some suitable open source DBMS technology. For example, MonetDB [2, 4, 5] seems to fit quite well for this purpose as its basic structure is the binary table. However, given that the vast majority of transactional databases and data warehouses today are based on the relational model, it is important to have a method of mapping the context model to the relational model. In this section we present such a method, comprising three mappings as follows :

- Mapping of a context to a relational star schema
- Mapping of a path expression to a relational expression
- Mapping of an OLAP query to an SQL query

Mapping a context to a relational star schema
Given a functional schema S, there are several ways to map it into a relational schema $rel(S)$. The simplest way is to represent each arrow of S by a binary table, and define the set of all binary tables to be the schema $rel(S)$. However, the evaluation of OLAP queries will then require the frequent use of joins. A more efficient mapping is the one that produces a so called *star schema*, whose tables and constraints are defined as follows :

Tables :

- Define a table FT containing the origin of S and all its immediate successors as its attributes. Call these immediate successors the *base attributes*, and call the table FT the *fact table*.
- For each base attribute B, if there is at least one successor of B, define a table BT containing B and all its descendants as attributes. Call this table the *B-table*.

Constraints :

1. In each of the tables defined above, any arrow of S connecting two of its attributes becomes a functional dependency of that table (hence it might be that some tables are not in BCNF).

2. There is a foreign key dependency from the fact table to every other table BT: $\pi_B(TF) \subseteq \pi_B(BT)$.

Example :
The star schema for our running example is the following (underlined attributes form the key of each table) :

$FT(\underline{Date, Store, Product}, Num)$
$DateT(\underline{Date}, Month)$, with $\pi_{Date}(FT) \subseteq \pi_{Date}(DateT)$
$StoreT(\underline{Store}, City, Region)$, with $\pi_{Store}(FT) \subseteq \pi_{Store}(StoreT)$
$ProductT(\underline{Product}, Category, Supplier)$, with $\pi_{Product}(FT) \subseteq \pi_{Product}(ProductT)$

Note that the table $StoreT$ is not in Boyce-Codd Normal Form, and that the table $DateT$ is not necessary to store (as the month can be determined from the date).

Mapping a path expression to a relational expression
The following algorithm maps a path expression e over S to a relational expression $rel(e)$ over the star schema $rel(S)$:

if the target of e contains only base attributes
then $rel(e) = \pi_{target(e)}(FT)$
else $rel(e) = \pi_{target(e)}(FT \bowtie T_1 \bowtie ... \bowtie T_k)$,
where $T_1, ..., T_k$ are all the tables each of which contains at least one non base attribute appearing in the target of e.

Example:
The path expressions :

$$e_1 = g \wedge (h_2 \circ h) \text{ and } e_2 = n$$

map to the following relational expressions :

$$rel(e_1) = \pi_{Store,Sup}(FT \bowtie ProductT)$$
$$rel(e_2) = \pi_{Num}(FT).$$

Mapping an OLAP query to an SQL query
To map an OLAP query $Q =< c, m, op >$ to a relational query $rel(Q)$ it is sufficient to replace the path expressions c and m by $rel(c)$ and $rel(m)$, to obtain

$$rel(Q) =< rel(c), rel(m), op >.$$

The query $rel(Q)$ is then evaluated using a "group by" instruction. This is possible if one observes that the "group by" instruction of SQL simply computes inverses of a special kind of functions, namely projections.

Example :

$$Q =< g \wedge (h_2 \circ h), n, sum >$$

maps to

$$rel(Q) =< \pi_{Store,Sup}(FT \bowtie ProductT), \pi_{Num}(FT), sum >,$$

and $rel(Q)$ is evaluated as follows:

select $Store, Sup, sum(Num)$ as $TotNum$
from $join(FT, ProductT)$
group by $(Store, Sup)$

The previous SQL instruction computes the inverse of the projection function $\pi_{Store,Sup}$ thus creating a partition of the table $join(FT, ProductT)$ into sub-tables; then in each sub-table T, it applies the function n to each tuple of the sub-table to find the corresponding number; and finally sums up the numbers found to return the total number for each sub-table.

As a final remark, in multidimensional database parlance, the determining base attributes (such as Date, Store, Product in our running example) are called "dimensions", or "categorical attributes", while the determined base attributes (such as Number in our running example) are called "measures", "numerical attributes", or even "summary attributes" [1, 7]. However, there is no formal way of deciding which attributes should be dimensions and which should be measures. This choice is left as a database design decision.

5. A USER-FRIENDLY INTERFACE

An interface is currently under implementation whose gross architecture is shown in Figure 7. The interface performs two main tasks. First, it allows the designer to input a pair (U, F), visualize a quotient graph, select a context and get as output a star schema, under which is stored the data to be analyzed. This task is performed using the algorithm presented in the previous section. Once the star schema is defined data can be entered either through the interface or directly under the star schema. Second, it allows the analyst to formulate OLAP queries against the star schema using as an interface the context schema from which the star schema was defined.

The interface presents the context schema to the user in one of two forms: either as a graph with clickable nodes or as an indented list of clickable items. To formulate a query $Q =< c, m, op >$, the user simply clicks on nodes of the schema (or items of the indented list) and selects an operation from a popup menu. Once the query Q is defined through clicks, the interface passes it to the query mapper which maps Q to an SQL query and passes it over to the relational engine (see Figure 7). Finally, the relational engine processes the SQL query and passes the answer to the interface. More precisely, during formulation of the query $Q =< c, m, op >$, the path expressions c, m, and the operation op are defined in a sequence of three modes, as follows:

Classifier mode

The user defines a path by clicking on one or more nodes of the schema until a single path from the origin can be determined. If the schema is a tree, then a single click is sufficient to specify a path, otherwise two or more clicks might be necessary in order to specify a single path from the origin. This process is repeated as many times as there are paths that the user wishes to define. Upon exit from this mode, the interface defines c by composing along each path defined by the user, then pairing all compositions. For example, in Figure 6, as the schema is a tree, it is sufficient to click on a single node to define a path from the origin. Thus if the user clicks first on *Region* and then on *Supplier*, the system will generate the path expression $c = (g_2 \circ g_1 \circ g) \wedge (h_2 \circ h)$.

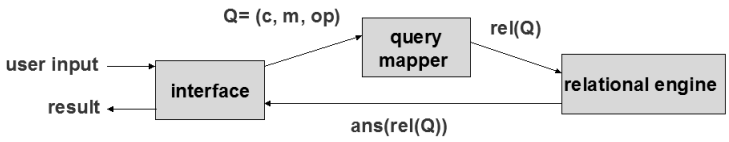

Figure 7: User-friendly interface prototype.

When the user indicates the end of classifier mode, the interface switches to the measurement mode.

Measurement mode

The measure m is defined in exactly the same way as c is defined. The difference here is that, upon exit from this mode, the interface "infers" the operations applicable on the target of m and puts them in a popup menu. For example, assume in Figure 6 the user specifies n as measure. Since the domain of $Number$ is the set of integers, the system will put in the popup menu all operations that are allowed on integers (sum, max, min, avg, $count$, etc.). When the user indicates the end of measurement mode, the interface switches to the aggregation mode.

Aggregation mode

The interface presents to the user the popup menu created during the measurement mode. The user then selects one operation op by clicking on its name in the popup menu. Upon exit from this mode, the interface defines the query $Q =< c, m, op >$ and passes it to the mapper (which maps it to an SQL query and passes it to the relational engine for evaluation). After evaluation, the relational engine passes the answer to the interface for exploration by the user.

In its current form, the interface returns the answer to an OLAP query using the tools available at the relational engine, namely, the answer is returned either in tabular form or in one of the available visualization modes (under the control of the user).

Regarding loading of data, this is also done through the interface: by clicking on an arrow of the schema, the user can enter a set of pairs conforming to the definition of that arrow; a check of "functionality" is performed and a message is returned to the user ("error", if the functionality of the arrow is violated, and "accepted" otherwise). The data entered is periodically propagated to the data warehouse.

6. CONCLUDING REMARKS

We have presented an approach to data analysis based on functional dependencies. Given a set U of attributes and a set F of functional dependencies, our approach relies on a few basic notions of relational database theory to define the notion of analysis context. Our main contributions are (a) an algorithm for generating all analysis contexts embodied in the pair (U, F), (b) a functional language in which analysis queries can be formulated within a context, (c) a formal approach to query optimization, and (d) an algorithm for mapping a context and its associated functional language to a relational star schema; the mapping is fully automatic and forms the basis for an interface currently under development that allows the designer to enter a pair (U, F), visualize the quotient graph, select a context and get as output a star schema. We note that the input pair (U, F) might be the result of semantic reconciliation between attributes and between dependencies coming from two or more different (transactional) databases.
Future work aims at the implementation of our model based on some open source solution. In this respect, we are currently studying the possibility of using MonetDB [2, 4, 5], which is an open source database system for high performance applications in several areas, including data analysis.

7. REFERENCES

[1] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. pages 232–243, 1997.

[2] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, October 1999.

[3] P. Buneman and R. E. Frankel. Fql: a functional query language. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 52–58, New York, NY, USA, 1979. ACM.

[4] M. Ivanova, M. L. Kersten, and N. Nes. Adaptive segmentation for scientific databases. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 1412–1414, 2008.

[5] M. Ivanova, M. L. Kersten, and N. Nes. Self-organizing strategies for a column-store database. In *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008*, volume 261, pages 157–168, 2008.

[6] R. Ramakrishnan and J. Gehrke. *Database Management Systems (Third edition)*. McGraw-Hill, 2002.

[7] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical report, IBM Almaden Research Center, San Jose, CA, 1996.

[8] N. Spyratos. A functional model for data analysis. In *Flexible Query Answering Systems, 7th International Conference, FQAS 2006, Milan, Italy, June 7-10, 2006, Proceedings*, pages 51–64. Springer, 2006.