EXLEngine: executable schema mappings for statistical data processing

Paolo Atzeni
Università Roma Tre
atzeni@dia.uniroma3.it

Luigi Bellomarini
Bank of Italy
luigi.bellomarini@bancaditalia.it

Francesca Bugiotti
Università Roma Tre
franbugiotti@yahoo.it

ABSTRACT

Data processing is the core of any statistical information system. Statisticians are interested in specifying transformations and manipulations of data at a high level, in terms of entities of statistical models such as time series. We illustrate here an experience at the Bank of Italy where (i) a language, EXL, has been defined for the declarative specification of statistical programs, (ii) an approach for the translation of EXL code into executables in various target systems has been developed, and (iii) a concrete implementation, EXLEngine, has been carried out. The approach leverages on schema mappings as an intermediate specification step, in order to facilitate the translation from EXL towards several target systems.

Keywords

Schema mappings, statistical data, data warehouse, ETL

1. INTRODUCTION

Data processing is of primary importance in statistical information systems of major organizations, since it allows for the generation of statistical products, which are the finished, or at least deliverable, artifacts of the whole process of statistical data treatment [8]. A common statistic production flow [17] starts from the elementary (raw) data, acquired in many forms from the external environment. Elementary data are provided in a number of formats and fed into the system in an initial phase of the process generally known as *collection*. Collected raw data are then processed in order to obtain aggregated or elaborated information, valuable for the decision process relying on the statistical information system. This latter phase, known as *statistical data production*, precedes the *dissemination* which involves all the activities that concretely package and deliver products to the stakeholders [11]. In the area of automatic processing of statistical information, a major goal would be the integration of the three phases mentioned above within a unique information system.

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy

In the recent literature, a lot of consideration has been devoted to the approaches that try to provide support to transformations in various contexts, by means of algebras and languages for querying, manipulating, extracting and transforming data. In this respect, schema mappings are considered a powerful formal device to describe transformation and processing of data [6, 16] . ETL (Extract - Transform - Load) tools, commonly used in data warehousing, can be seen as application counterparts of mapping theory; the close relationship between schema mappings and ETL executable flows was practically clarified in the Orchid prototype [10] but had been consciously understood since Clio early studies [15]. It is also worth mentioning that while most of the mappings theory has been developed for the relational model, arguments for more general, model-independent approaches have been made [2, 4, 6].

As a matter of fact, there are several kinds of engines used to run or support statistical programs, which include domain-independent ones, such as relational databases of ETL tools, as well as specific ones, such as R or Matlab. Therefore, the need for a unifying approach that supports several implementation platforms has clearly emerged. The Bank of Italy has sponsored these objectives, playing for decades a key role in the process of standardization and devising Matrix [9], a statistical data model, actually adopted in the Bank, which falls in the class of SDMX[1] (Statistical Data and Metadata Exchange), the internationally adopted model which benefits from these efforts. The Bank of Italy also devised EXL [9] (EXpression Language), a specification language for statistical programs over cubes (involving sum, difference, aggregations of cubes etc.).

In this paper, we illustrate an experience carried out at the Bank of Italy, which leverages the expressive power of schema mappings and the huge expertise on them present in the IT community, to build a theoretical and practical bridge between the abstract specification of a statistical program (in EXL) and its software execution (in relational databases, statistical languages and ETL tools).

Our conceptual goal is to formalize the link between programs expressed in terms of statistical models and their executable form by means of schema mappings. We show that the result of the execution of the declarative schema mappings corresponds to the algorithmic application of program expressions. To achieve this target, we formulate a data ex-

[1] http://sdmx.org/

change problem instance out of the generated schema mappings and show that its solution is equivalent to the effect of the statistical program.

Our practical goal is to provide a technological hub to decouple the abstract specification of statistical procedures from their technical implementation. The translation of programs into schema mappings allows for their execution in a number of target systems.

We illustrate our concrete contribution through *EXLEngine*, the engineered system actually adopted by the Bank of Italy for the generation and execution of schema mappings out of declarative statistical programs. Programs written by Bank statisticians are fed into EXLEngine. They are then translated into a set of schema mappings. Finally, the produced mappings are translated into an executable form according to the specific target system. For example, mappings (and so the original program) could be translated into SQL to delegate the execution of the program to a DBMS; into ETL jobs to pass the execution to a specialized stream-like architecture such as an off-the-shelf ETL engine; into a specialized language for a statistical tool (such as R, FAME, Matlab, etc.).

In the next section we give an overview of our approach and at the end of it we briefly illustrate the organization of the subsequent sections.

2. OVERVIEW

In this section, we illustrate our approach by using an example, a small statistical program, for which we first show the specification in EXL, then the equivalent schema mapping and finally portions of its implementations in various target systems.

So, let us consider the following EXL program, which calculates the percentage change of the GDP (Gross Domestic Product) trend by quarter given the GDP per capita by region and quarter and the population of each region by day.

```
(1) PQR := avg(PDR, group by
                    quarter(d), region)
(2) RGDP := RGDPPC * PQR
(3) GDP := sum(RGDP, group by quarter)
(4) GDPT := stl_T(GDP)
(5) PCHNG := (GDPT - shift(GDPT,1))
                     * 100 / GDPT
```

Entities denoted by uppercase strings are statistical functions (called *cubes*, as in data warehousing). They have arguments, which are omitted in the listing and we briefly describe now. $\mathrm{PDR}(d, r)$ represents the population of a region r at the end of the day d. $\mathrm{PQR}(q, r)$ represents the same population during a quarter, and it is calculated (line 1) from PDR by changing its sampling frequency from day to quarter and aggregating the population measure with an average function. $\mathrm{RGDP}(q, r)$ is the regional gross product, which is obtained (line 2) as product of RGDPPC (regional gross domestic product per capita) by quarter and the (previously calculated) average population in the same quarter. $\mathrm{GDP}(q)$ is then obtained (line 3) as the sum over regions of $\mathrm{RGDP}(q, r)$. Then, in line 4, the *seasonal decomposition*[2] operator stl_T is used to isolate the trend $\mathrm{GDPT}(q)$. Finally (line 5) the PCHNG is obtained computing the difference between the values of GDPT in two consecutive quarters, multiplying by 100 (to have a percentage) and dividing by the trend itself.

As we will illustrate in Section 4, we can reformulate every statistical program in terms of a schema mapping, with a set of tgds[3] over relations that correspond to cubes. For every cube C, we define a relational symbol C', which has the same arguments as the cube, plus one, corresponding to the value of the function. Indeed, in the following we will often use the same symbol for the cube in EXL and for the relational symbol in tgds, as it will be clear from the context what we refer to. So, for example, for function $\mathrm{PDR}(d, r)$ (the population of a region r at the end of the day d), we have a relation $\mathrm{PDR}(d, r, p)$, whose tuples include also the value p for the population. The tgds we need are extensions of those commonly used in data exchange settings, because we need to represent operators. In the example, we would have the following[4]

(1) $\mathrm{PDR}(t, r, p) \rightarrow \mathrm{PQR}(\mathrm{quarter}(t), r, \mathrm{avg}(p))$
(2) $\mathrm{PQR}(q, r, p) \wedge \mathrm{RGDPPC}(q, r, g) \rightarrow \mathrm{RGDP}(q, r, p * g)$
(3) $\mathrm{RGDP}(q, r, g) \rightarrow \mathrm{GDP}(q, \mathrm{sum}(g))$
(4) $\mathrm{GDP} \rightarrow \mathrm{GDPT}(\mathrm{stl\_T}(\mathrm{GDP}))$
(5) $\mathrm{GDPT}(q, r_1) \wedge \mathrm{GDPT}(q-1, r_2) \rightarrow \mathrm{PCHNG}(q, (r_1 - r_2) \times 100/r_1)$

The extensions with respect to usual tgds do require some care, even in the definition, so we assume for now the intuition behind them, and we will explain them later. This especially true for complex operators such as the seasonal decomposition, where each tuple is a function of all tuples of the operand; for this reason we use no variables in tgd (4).

The schema mapping represents in our approach an intermediate, implementation-independent step, which is then translated into an executable version in a target system. Many translations are indeed possible and are supported. We have translations into SQL, R and Matlab.

For example, the translation in SQL for the calculation of PCHNG (tgd (5)) can be implemented by means of the following statement.

```
INSERT INTO PCHNG(Q,R)
SELECT G1.Q, (G1.R - G2.R) * 100 / G1.R
FROM GDPT G1, GDPT2 G2
WHERE G1.Q = G2.Q - 1
```

[2]The seasonal decomposition is an operator that decomposes a time series into various components, one of which is the *trend*, which, roughly speaking considers medium- or long-term movements, ignoring seasonal, cyclic (and stochastic) ones [7, 20].

[3]As we will see in Section 4, we also have some egds, which enforce the functional nature of cubes, but we can ignore them here, given that their satisfaction is always guaranteed, as we will argue.

[4]Following common practice, we omit universal quantifiers. Also, given that we need only total tgds, wee have no quantifiers at all.

Other possible target systems and languages include the ones that are specifically designed for statistical elaborations, such as R or Matlab. They are typically vector or matrix oriented and offer a number of powerful statistical functions. In most cases, the translation from a tgd into such target languages is intuitive.

For instance, consider tgd (4), which calculates GDPT out of GDP and its translation in R. This requires the specification of the various components of the time series, by means of function stl(), followed by the extraction of the trend component, as follows.

```
GDPC=stl(GDP,"periodic")
GDPDT=GDPC$time.series[ ,"trend"]
```

The same tgd would be translated differently, but in a straightforward way, if we assume a trend isolating library in Matlab, acting on vectors.

```
GDPC=isolateTrend(GDP)
```

A possible perspective on statistical data production, which will be discussed in Section 5, is ETL. A statistical program can be intuitively seen as an ETL job composed of a number of flows each representing a tgd statement. All flows have the same structure and involve: data source steps, feeding data into the ETL stream; merge steps, combining streams coming from different sources; calculation steps, performing simple or user defined algebraic or statistical calculations; output steps, writing the results back into the system. For each relation in the lhs there is a data source step in the flow. Data streams coming from these steps are merged on the basis of dimensions, while their measures are combined with the calculation step.

The approach we have sketched above is indeed adopted in *EXLEngine*, the engineered system actually adopted in the Bank of Italy for the generation and execution of schema mappings out of declarative statistical programs. Programs written by Bank statisticians are fed into EXLEngine, which translates them and generates executables in the various target systems.

The remainder of the paper is organized as follows. In Section 3 we present a short overview of EXL, a specification language for statistical programs. In Section 4 we present the approach adopted in EXLEngine to generate schema mappings out of statistical programs. We address correctness issues by recognizing a data exchange setting from the generated mappings. In Section 5 we show how EXLEngine can translate the executable schema mappings into various executable forms, such as SQL queries or ETL jobs. The architecture of EXLEngine is then briefly presented in Section 6. In Section 7, we discuss related work and in Section 8 we draw our conclusions.

3. THE EXPRESSION LANGUAGE

In this section we illustrate the model we adopt for the representation of statistical data and the language we use for the high-level specification of the transformations of interest.

As we saw in the introduction, we will show in the subsequent sections how such a specification can be translated into executable programs.

The model and language have been defined with the goal of being closer to the actual activities of statisticians and independent of the specific implementation.

The data of interest in this context can be modeled by means of *dimensional cubes*, similar to those used in data warehousing activities, and the associated transformation processes can be modeled by *statistical programs* acting on the cubes. Let us formalize these concepts, first the model and then the language for transformations.

We define a *cube* (as usual) as a partial function

$$F : X_1 \times \ldots \times X_n \to Y$$

where F is the *identifier* of the cube, and $X_1, \ldots, X_n$ (the *dimensions*) and Y (the *measure*[5]) are all sets. As in databases (relational and other) we associate a name with each dimension (and call it *dimension* indeed), so that we can effectively refer to it. So, we will also write the cube as $F(D_1, \ldots, D_n)$ where $D_1, \ldots, D_n$ are the dimensions. The function is defined over tuples of the form $(x_1, \ldots, x_n)$, where $x_1 \in X_1$, $\ldots$, $x_n \in X_n$, associating a value (a measure) y in Y with each *dimensions tuple* $(x_1, \ldots, x_n)$ (on which the function is defined; there are as well tuples on which the function is not defined—the function is sparse). Given the frequent implementation of cubes on relational databases, it is common to refer to the $(n+1)$-tuple $(x_1, \ldots, x_n, y)$ as a *cube tuple*.

In the specific practice of statistics, it is often convenient to distinguish dimensions related to time from the others, and to give specific attention to some extreme cases. In particular, if a function has only one variable that corresponds to a time dimension, then it is named *time series*; in our approach, time series are also treated as cubes. For this reason we would need to be specific on the nature of the domains for the dimensions. In the following, for the sake of simplicity, we will mainly ignore types in the discussion, mentioning them only when really needed. For the same reasons, we assume that measures are all numeric.

Let us now turn our attention to *EXL (EXpression Language)* a specification language for statistical programs over cubes, defined and adopted by the Bank of Italy. We have already seen an example in Section 2. An EXL statistical program is a sequence of *statements*. A statement is an assignment where the left-hand side (lhs) is a cube identifier and the right-hand side (rhs) is an *expression*.

In an EXL program, cube identifiers are partitioned into two categories: *elementary*, whose tuples are available as base data provided to the system, and *derived*, defined by means of expressions. This partitioning is similar to the one in relational databases between *base tables* and *views*, where each view has a unique definition, and to the one in deductive databases, and in Datalog specifically, between *extensional* and *intensional* predicates. In an EXL program, the ex-

[5]In general, it could be possible to define cubes with more measures, but this would not add much to the discussion.

pression that specifies the definition of a derived cube in a statement may contain only elementary cubes and other derived ones specified in previous statements in the program. So, no recursive definition of derived cubes is allowed, as this is not needed in the statistical applications of interest. Moreover, as a cube defines a function, there needs to be a unique way to obtain it, and so a cube identifier must not appear as lhs more than once (as opposed to what happens in Datalog, where intensional predicates can be defined by means of multiple rules, because there is no functional restriction).

Expressions specify how the tuples in a derived cube are calculated. They can be recursively defined as follows:

- a cube identifier (e.g. C) is an expression; we use the term *cube literal* for this base case; the type of the expression is the same as that of the cube;
- the application of any n-ary EXL operator to n expressions (whose types are compatible with the operator) is an expression; its type is determined by the specific operator.

Let us now discuss operators. The language has many of them and comprises elementary algebraic ones (sum, product, etc.) as well as all the complex operators commonly adopted for statistical analysis (including linear regression, seasonal decomposition, aggregations such as average, median, standard deviation). Given that each operator has a specific syntax and semantics and that a complete generalization is not possible (nor a detailed presentation of all of them), we illustrate now some interesting operators, which represent the main categories, with the associated variants and difficulties.

The common feature of operators is that, obviously, an operator produces a result cube (a function with at most one value for each dimension tuple) from one or more input cubes. In general, the value of the cube on a dimension tuple may depend on the values of several tuples (this is for example the case in aggregation operands and in many statistical ones). In this respect, we distinguish two main classes, as follows. We say that an operator is *tuple-level* if a value in the result depends only on the value of at most one tuple for each of the operands, while we say it is *multi-tuple* if a value of the result depends on a (non-singleton) set of tuples of an operand.

As it is common in many languages, we have a syntax (with special symbols) for some algebraic operators and a function notation (with an identifier and operands in parenthesis) for the others.

Operators of all kinds might have, beside operands (which are in turn expressions), also additional arguments, which can be scalar parameters or structural elements. Examples of scalar parameters include the logarithm base, as in $\log(2, \mathrm{e1} * 3)$, or the shift in the time series we already saw in the example in Section 2. Structural elements arise in all contexts where the result cube is a result of a restructuring of the operand, as in the cases of aggregation we discuss below.

Tuple-level operators are *scalar* or *vectorial*. Scalar operators have one cube operand and scalar parameters (usually one). They include the natural ones on the measures of cubes (sum, subtraction, product, division with a constant, increment, logarithm, exponential, trigonometric function). Here the expression (the resulting cube) has the same dimensions as the operand, and it is defined (with usual the semantics for each operator) on the dimension tuples on which the operand has a value for which the operator is meaningful: for example, given the expression e, we have that the expression $1/e$ is defined on all dimension tuples on which the value of the cube denoted by e is not zero.[6] Other scalar operators transform the dimensions. The most common here is the *shift*, which is essentially a sum on the values of a numeric dimension or (with a suitable, but natural definition) on a time dimension. The semantics of the time shift with parameter s is that the result cube is defined on dimension values $t + s$ for each dimension value t on which the operand cube is defined,[7] and with the same value: given expression e, we have that $\mathrm{shift}(e(t + s)) = e(t)$, for all t.

Vectorial operators have two (or even more, but this is not essential here) cubes, generating a third one as a result. The two operands and the result as well have the same dimensions[8] (same name and type for each), and the semantics is defined as expected, for each dimension tuple. A nontrivial issue is how to deal with cubes that have the same dimensions but their values exist on different dimension tuples: here different versions exist, we mainly refer the simplest, which produces the result cube tuple only for dimension tuples that appear in both cubes, but there are others assuming a default value for the "missing" tuples (example, in the sum operator, we could have zero as the default value).

The second class, that of tuple-level operators, includes many of them, which are indeed important in the production of statistical data, as they restructure cubes, calculating new values from sets of previous ones. Among them, we have a significant subclass whose elements can be considered as black boxes, as they receive one cube in input and transform it by producing another cube. They have no additional parameters or clauses and so their semantics is just defined by the black box function they refer to. An interesting representative is the seasonal decomposition (`stl`) operator we mentioned in Section 2. Another specific, widely used subclass is that of aggregation (or summarization) operators, which "roll up" cubes, by applying a specific arithmetic operator (for example sum, max, min, or average) to the values of the cube that correspond to dimensions with the same value. Here the syntax is the following

aggr(*e*, `group by` *dimensionList*)

where *aggr* is one of the aggregation operators and *dimensionList* is a list of dimensions in e or scalar expressions over

[6] We said that measures are numeric. Indeed, we should distinguish on the basis of the specific type, but we omit this discussion.

[7] The example refers to a time series, with just one dimension. The generalization to a cube with more dimensions with a shift applied to a time one is a bit intricate in notation but straightforward.

[8] There are indeed versions that operate on cubes with different dimensions, but they are not much relevant here.

them (for example, the application of the `quarter` function to a date dimension, as we saw in statement (1) in the example in the Overview section). The semantics is essentially the same as we have in SQL aggregate queries: the result cube contains only the dimensions in *dimensionList* and it is defined as follows: let $(x_1, \ldots, x_k)$ be a tuple with one value for each dimension in *dimensionList* and V be the bag[9] of values in cube e that are associated with dimension tuples (in e) that coincide with $(x_1, \ldots, x_k)$ on *dimensionList*. Then, the value of the result cube on $(x_1, \ldots, x_k)$ is the result of applying function *aggr* to the bag V. The cube tuple exists only if the bag V is non-empty.

To complete the discussion on the semantics of EXL, let us say that, as it is easy to guess, a statement assigns the result of the expression in its rhs to the cube in its lhs.

4. GENERATING SCHEMA MAPPINGS FROM STATISTICAL PROGRAMS

Let us now see how a schema mapping can be generated out of an EXL statistical program. This is illustrated in Subsection 4.1 (and implemented in our tool EXLEngine). Then, in Subsection 4.2 we argue for the correctness of the translation, showing that a solution to the data exchange problem specified by the mappings equals the result of the application of the statistical program.

4.1 The generation of schema mappings

With respect to the common theory of mappings [16], we need to make some extensions to the well known language for dependencies. In fact, as we saw in Section 3, we do need to handle operators, which can be complex, with results that depend on sets of input tuples, as in the case of aggregation functions or most statistical operators. So, we will need to provide suitable definitions for the semantics of the dependency language as well as on the chase procedure for correctness proof.

As usual, the mapping M we build has the form $M = (S, T, \Sigma_{st}, \Sigma_t)$. Where S and T are the source and target schemas, respectively, Σ_{st} and Σ_t are the source-to-target and target dependencies, respectively.

The source relational schema S contains a relational symbol $F_i(X_{i,1}, \ldots, X_{i,n_i}, Y_i)$ for each cube in the EXL program of interest.[10]

The target schema T contains the same relations, which we however need to rename, since we assume (as usual [14]) $S \cap T = \emptyset$. So, for each $F_{S,i}(X_{i,1}, \ldots, X_{i,n_i}, Y_i) \in S$ we have a relational symbol $F_{T,i}(X_{i,1}, \ldots, X_{i,n_i}, Y_i) \in T$, and a *source-to-target tuple generating dependency (tgd)* in Σ_{st} that "copies" the source relation into the target one:

$$F_{S,i}(x_1, \ldots, x_{n_i}, y) \rightarrow F_{T,i}(x_1, \ldots, x_{n_i}, y)$$

However, we will not refer to these tgd any longer in the rest of the paper, as their role is straightforward, and we will keep on using the same symbol for the relation in the source and its copy in the target.

An additional auxiliary set of dependencies is needed on the target, on the basis of the fact that cubes represent functions, with one value for the measure for each dimension tuple. Indeed, in each cube F_i in the target schema we have a functional dependency from the dimensions to the measure. This is modeled by means of an *equality generating dependency (egd)*, of the form:

$$F_i(x_1, \ldots, x_{n_i}, y_1) \wedge F_i(x_1, \ldots, x_{n_i}, y_2) \rightarrow (y_1 = y_2)$$

Then, we have dependencies that correspond to the EXL statements. For each statement, we have one or more target tgds, depending on the expression in the rhs of the statement. For the sake of simplicity, we assume here that the expressions in EXL statements include one operator,[11] and so there will be only one tgd for each statement. This assumption does not cause any loss of generality in the expressive power, as we could add additional statements and auxiliary cubes to handle intermediate results. For example, statement (5) in the example in Section 2 could be replaced by three statements, as follows:

```
(5a) GDPT1 := shift(GDPT,1)
(5b) CHNG := GDPT - GDPT1
(5c) RCHNG := CHNG / GDPT
(5d) PCHNG := 100 * RCHNG
```

It should be noted that, in this way, we obtain many tgds, each with one operator. Indeed, in practice, our tool is able to simplify them, as it was shown in the example in Section 2, where statement (5) generates the single tgd (5), with a complex expression.

According to these hypotheses, all EXL statements have the form

- C `:=` *op* `(`C_1`, ..., `C_k`)`

where the operator *op* might have the special syntax or the standard function notation, and the number of operands would depend on the operator itself.

Let us first see tgds for tuple-level operators, where, as we saw, the result cube is computed value by value, by applying a function to the individual values of the input cubes for one dimension tuple for each of the operands. In general, the dimension tuple in the result need not be the same as in the operand(s): for example, in the time shift operator, the value in the result cube is the same as in the operand, but for a different dimension tuple. In all these cases, we can find an extension of the usual notion of tgd, with as many atoms in the lhs as the number of operands and some scalar expression in the rhs, for the measure or for one of the dimensions. For example, consider the following expressions, a scalar multiplication, a vectorial sum, and a time shift

[9] That is, repeated elements are meaningful.

[10] As anticipated, we blur the distinction between a cube and the respective relation; F_i denotes both the cube and the corresponding relation.

[11] We could say at most one operator, but it is easy to assume that there are no statements that just copy a cube with no additional operations.

- `C2 := 3 * C1`
- `C5 := C3 + C4`
- `C7 := shift(C6,1)`

They would give rise to the following tgds (assuming the cubes in the first two statements all have two dimensions and those in the third have only one):

- $C_1(x_1, x_2, y) \rightarrow C_2(x_1, x_2, 3 \times y)$
- $C_3(x_1, x_2, y_1), C_4(x_1, x_2, y_2) \rightarrow C_5(x_1, x_2, y_1 + y_2)$
- $C_6(t, y) \rightarrow C_7(t - 1, y)$

These tgds are indeed a bit more complex than those usually found in data exchange settings, but their semantic is a straightforward extension of the classical one, in the sense that a tuple has to exist in the relation in the rhs for each tuple in the lhs (for unary operators, or pair of tuples for n-ary operators and so on). Also, these are full tgds, as there are no existentially quantified variables in the rhs. Therefore, the values in the generated tuples are uniquely defined, both when they are copied and when they are calculated.

Let us now consider multi-tuple operators, which produce values that are calculated from sets of tuples, usually within a single cube. These include all the aggregation operators as well as many interesting statistical ones. Here tgds require special care, because the constraints they specify need to refer to a relation as a whole, rather than to individual tuples independently from one another. For example, statement (3) in the example in Section 2 specifies the sum of different values of the measure, one for each of the tuples that refer to a given quarter. This means that, in the extreme case, a value for the result cube could even depend on all the tuples of the input cube.

Here, given a statement of the form

- `C2 := aggr ( C1 , group by D1, ..., Dk )`

assuming that $D_1, ..., D_k$ are the first k dimensions of C_1, we would have the following tgd:

- $C_1(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n, y) \rightarrow C_2(x_1, \ldots, x_k, aggr(y))$

whose semantics would be:

- for every different tuple $x_1, \ldots, x_k$ in the projection of C_1 on $D_1, ..., D_k$, there is a tuple $x_1, \ldots, x_k, y'$ in C_2 (so with the same $x_1, \ldots, x_k$ values for the dimensions) with a value y' for the measure that is the result of the aggregation function *aggr* applied to the multiset of values that the measure has in the tuples of C_1 that coincide with this tuple on $D_1, ..., D_k$.

It is worth mentioning that aggregation functions have been introduced in various settings that make use of logic formalisms, and the need for a careful definition of semantics arose in all of them, especially when a procedural semantics is added (for efficiency of evaluation) to a model based one. A simple solution, which would be sufficient for our goals, is based on stable model semantics or on stratified semantics [19], where the basic idea for our case would be very simple: an aggregation function is computed only when its input operands are completely known. Given that we have no recursion, this becomes easy to achieve in our case, following the total order in EXL statements.

In the Overview (Section 2) we have already shown a set of tgds, which have been derived according to the procedure we have just described, with one minor extension: we had two statements, (1) and (5) with more than one operator, and we have generated a tgd by means of a combination of the steps just discussed. In statement (1) we have a scalar function (quarter) that operates on the values of a dimension, and we have it in the same way in the tgd. In statement (5), we have four operators, and the resulting tgd takes care of all of them in a straightforward way.

4.2 Correctness of the schema mappings

Let us now argue for the correctness of the schema mappings generated out of the EXL programs as illustrated in Subsection 4.1.

Thus, let us consider the data exchange problem associated with the schema mapping $M = (S, T, \Sigma_{st}, \Sigma_t)$: given M and a finite instance I of S, find a finite instance J of T such that $\langle I, J \rangle$ satisfies Σ_{st} and J satisfies Σ_t. Instance I is a collection of facts for the S atoms representing the cubes, and so it coincides with the input of the EXL program.

We prove that this data exchange problem always has a solution, which can be found by means of (a suitable variation of) the chase. Then, we show that the chase indeed generates the same instance of the target schema as the EXL program, and so the two are equivalent and the solution coincides with the output of the EXL program.

The chase terminates, succeeds and solves the data exchange problem

An EXL program with an instance I as input always terminates and returns an output; the program is acyclic, because each cube is defined by means of a statement, and makes use of other cubes which are available. Each statement involves operators which are well defined as functions and so each of them terminates.

The existence of solutions to the data exchange problem is usually proved by means of the *chase* procedure [13, 16], which "applies" constraints to an instance, "forcing" their satisfaction (for tgds this means adding new tuples corresponding to the rhs for each set of tuples that unify with the lhs; for egds, this means equating values, and this may lead to failure, if there is need to equate constants). The procedure has a running instance of $\langle S, T \rangle$ which is initialized as $\langle I, \emptyset \rangle$ (the input instance I for the source schema and the empty instance for the target one). Then the target instance is modified by applying all the dependencies in Σ_{st} and Σ_t as long as they are applicable. Application of tgds means generation of new tuples in the target instance (as the name "tuple generating" suggests), while application of egds leads to modification to the values, if they violate the

dependency and they are not constants. Violations of egds that involve constants cause a "failure" of the procedure. In a classical data exchange setting, if the process terminates and does not fail, then the resulting instance is a solution to the data exchange problem,[12] which is then shown to exist.

Now, the data exchange problem we have in our case, as defined by the schema mappings shown in Subsection 4.1, exhibits some differences with respect to the classic case. We devise a variation of the classical chase, which always terminates for our data exchange setting.

The involved dependencies only contain *full tgds*, therefore there are no existentially quantified variable, and so all tuples are generated with constants. In the classical case (without aggregation) this condition is sufficient to guarantee termination [5, 13], because of the finiteness of the source instance, as the number of symbols in the target instance is bounded. Our tgds are also acyclic, and this is another sufficient condition for termination in the classical case.

However our tgds are indeed more complex than the classical ones, because of the aggregations and multi-tuple operators; thus the mentioned conditions are not sufficient for termination and some considerations have to be made. The presence of tuple-level operators does not affect termination as tuples in the target instance are generated in a bounded way for all tuples that unify with the lhs. For aggregations and the other multi-tuple operators, termination is guaranteed by the adoption of a slightly modified version of the chase in which the order of rules is constrained. Rather than allowing the applications of chase rules in any order, we follow a stratified approach: we consider the total order on derived cubes that corresponds to their definition in the EXL program, and apply rules in that order, by completely applying the rules corresponding to one statement, before considering the next one.[13] Here again it is clear that when a rule is applied to a specific tuple (or sets thereof), then it will not have to be applied again, and so the number of applications is finite and the chase terminates.

As a consequence, if the chase does not not fail, then it produces a correct solution to the data exchange problem as it can be easily seen that the result of the application of rules is unique and equivalent to the result of any other terminating chase instance that respects the stratification order.

The only possibility of failure is the violation of an egd. In our setting, the egds we have are those that guarantee the functionality of cubes: a cube does not have two tuples with the same dimension tuple. We can prove this by checking that it is not possible for our egds to generate two distinct tuples with the same dimension tuple. First of all, as we defined EXL with only one statement for each derived cube, it follows that it suffices to show that no single tgd generates two possibly conflicting tuples. Indeed, this does not happen because all tdgs generate the measure value as a function on the basis of the values of dimensions. For example, for the sum of cubes, if we assume that the input predicates are indeed functional cubes, then a value for the result cube is defined for each dimension tuple, as the sum of the two values (each univocally defined, by hypothesis) for that dimensions tuple, and this value is unique. Similarly, the tgd for a multi-tuple function is defined to have one value (and only one) for each dimension tuple. We have not given details for the various statistical operators, but we assume that they are all defined in a functional way.

So, we have that the chase terminates, does not fail and generates a unique result. This result also satisfies all the constraints, as egds cannot be violated (as we argued above) and violations of tgds are eventually eliminated by applications of them. Therefore, the chase generates indeed a solution for the data exchange problem.

The schema mapping problem is equivalent to the EXL program
In order to prove that the schema mapping generated out of an EXL program is actually equivalent to it, we argue that the instance J that is the solution of the data exchange problem is equal to the output of the EXL program.

This holds if J contains a relational fact for each cube tuple generated by the EXL program and viceversa. We can claim that this is the case because of the way the tgds have been defined: for each EXL statement we have a tgd, and each tgd generates one tuple (and exactly one) for each tuple generated by the EXL expression. This can be proven, in tedious but straightforward way for each of the operators, in both classes, tuple-level and multi-tuple.

5. GENERATING EXECUTABLE CODE

The final step to fill the gap between EXL specification language and a working implementation is the translation of the schema mappings generated by the algorithm described in Section 4.1 into a form that can be executed on target systems.

As mentioned in the Overview, DBMSs are often adopted as statistical data can be successfully stored in tables. In other cases, domain-specific tools, allowing for a specialized treatment of statistical calculations by means of dedicated data structures and operators, are used. For this category, we focus on R and Matlab, two widespread matrix oriented tools. Other approaches comprise statistics within business intelligence processes and assimilate calculations in ETL flows.

We move from the basic observation that tgds in the generated schema mappings can be orderly applied independently of one another, as we have shown in Section 4.2. This implies that, without loss of generality, we can concentrate on the translation of each tgd into its executable form.

In fact, if an executable script generates all the tuples implied by a single tgd, then it is correct with respect to that dependency and corresponds to a self contained chase step for the whole data exchange setting. As a consequence, the execution according to the total order of tgds of every script,

[12]Indeed, it is a solution with very interesting properties, but this is not essential here.

[13]This total order is not strictly necessary, the only thing that is needed is that the rules that involve these general operators are applied only after their operands have been fully computed.

generates all the tuples needed to solve the exchange setting.

In all cases we proceed by showing examples, which are however sufficiently general to show how our approach proceeds. It is important to underline that a given tgd can be formally translated into all the executable versions, one for each target system. However, depending on the specific operators used in the rhs, the translation may be actually feasible or not. In fact, as it is obvious, it is not the case that all operators are natively supported by all systems.

5.1 SQL mappings

As we have mentioned in Section 4, tgds may have two basic structures as they include a tuple-level or multi-tuple operator.

For the first category, consider tgd (2) in the Overview, which is quite general, with two elements in the lhs, with shared variables, and a tuple level operator in the rhs:

(2) $\text{PQR}(q, r, p) \wedge \text{RGDPPC}(q, r, g) \rightarrow \text{RGDP}(q, r, p * g)$

Our tool generates out of it an SQL insertion statement of the following form

```
INSERT INTO RGDP(Q,R,P)
SELECT C2.Q AS Q, C2.R AS R, C1.P*C2.G AS P
FROM PQR C1, RGDPPC C2
WHERE C1.Q = C2.Q AND C1.R = C2.R
```

The conjunction of atoms in the lhs is turned into a join of the corresponding relations, with the equality conditions generated out of the repeated variables in the lhs. Measures are combined by means of a tuple-level operator. The script correctly generates all the tuples implied by the tgd, since they are uniquely identified by appropriate matches (join condition on `Q` and `R`) on dimensions. The tuple-level operator (here multiplication) can be any system (or user) defined stored function implementing any scalar function.

Tgds with multi-tuple operators require more care. Let us consider tgds (3) and (4) of the example in the Overview:

(3) $\text{RGDP}(q, r, g) \rightarrow \text{GDP}(q, \text{sum}(g))$
(4) $\text{GDP} \rightarrow \text{GDPT}(\text{stl\_T}(\text{GDP}))$

Indeed, the two cases need to be handled in different ways. Tgd (3) is essentially an aggregation absolutely similar to those common in SQL, and so an executable statement can be generated in a straightforward way, including a `GROUP BY` clause and the `SUM` aggregation function as follows.

```
INSERT INTO GDP(Q, G)
SELECT Q, SUM(G) as G
FROM RGDP
GROUP BY Q
```

Tgd (4), instead, is not an aggregation, and it does not reduce the cardinality by partitioning and combining tuples, but essentially computes a new table where each tuple depends on many (possibly all) of the tuples in the argument. This can be handled by means of a more general approach (which could also include aggregations as a special case, but this would make things more complex). This is based on an extended dialect of SQL where *tabular functions* are allowed (and this is indeed available in most commercial systems). Tabular functions take in input one or more tables and return another table whose tuples are obtained by an arbitrarily complex elaboration of the input tuples. Thus for tgd (4) the following script is generated.

```
INSERT INTO GDPT(Q,G)
SELECT Q,G
FROM STL_T(GDP)
```

In the script, `STL_T` denotes the application of a tabular function to the table GDP. In particular it returns the trend component for a given time series stored in a table with established naming (and types) conventions; it can be either a system provided API (and indeed many commercial systems have statistical add-ons) or a user-defined stored function.

5.2 Mappings in specialized languages

Let us start again with tgds with tuple level operators only.

Let us first refer to R, the programming language for statistical computing. Unlike SQL, it is matrix oriented, with structures that are called *data frames* and many ad hoc operations defined on them. Therefore, the implementation of tgds has to refer to data frames.

A translation of tgd (2) requires various statements.

```
tmp <- merge(PQR,RGDPPC,by=c("q","r"))
tmp$i <- tmp["p"] * tmp["g"]
TGDP <- tmp[-c("p","g")]
```

The first statement uses the R operator called *merge* to build a temporary matrix by joining PQR and RGDPPC on q and r. The second statement adds a new column i to `tmp`, computed as the element-wise product of p and g. Finally TGDP is obtained by removing columns p and g from the operand.

Matlab is another example of matrix oriented tool and translations are based on similar ideas, with differences essentially on syntax. Let us see how tgd (2) is expressed in this language as well.

```
tmp=join(PQR, 1:2, RGDPPC, 1:2)
tmp[;5]=tmp[ ; 3] .* tmp[ ; 4]
TGDP=[tmp[ ; 1] tmp[ ; 2] tmp[ ; 5]]
```

Here the Matlab *join* operator is used to build the temporary matrix `tmp`, as the composition of PQR and RGDPPC; its columns, in order, correspond to variables q, r, p and g in the tgd. Then, the second statement adds a new column, i (in position 5), again as the element-wise product of p and g (here denoted by positions 3 and 4, respectively). The last statement builds matrix TGDP as the composition of the dimension vectors (in the first two positions) and i.

With respect to multi-tuple operators, we can observe that both R and Matlab offer a number of statistical functions

over matrices, which directly correspond to the operators we have in EXL and so in tgds. Therefore, the translation is often direct, and we have already shown such cases in the Overview, for example for a tgd involving seasonal decomposition.

5.3 ETL translation

ETL and Business Intelligence tools are another common family of target system. The translation engine can indeed turn schema mappings into executable ETL jobs by feeding the metadata catalog of the specific tool. This objective is of course facilitated if the tool under examination provides software API to build its catalog. EXLEngine actually supports Pentaho Data Integration,[14] an open source ETL product which has the advantage of being completely metadata driven.

For every tgd, an ETL flow is generated by the engine. All flows are finally tailored into a more comprising job according to tgds total order.

For each atom in the lhs, a *data source step* appears in the flow. The operator in the rhs is translated into a cascade of ETL operators, first joining input streams coming from data source steps on the dimensions, and then combining the measures. The first operation is performed by a *merge step*, which, in turn, is implemented in different ways in the systems, according to the ETL design and the chosen access plan. The joined stream is then processed by a *calculation step*, which actually implements the rhs of the tgd. ETL systems provide a number of algebraic tuple-level features in this kind of steps, calculating stream columns from other columns. Calculation steps can be easily replaced by user-defined steps in order to extend the statistical capabilities of the system. When multi-tuple operators are present in the rhs, then the native or user-defined calculation step may be combined with an *aggregation step* if some grouping is needed. Finally, the stream is written back into the system with an *output step*.

With reference to tgd (2) in the Overview, the ETL flow in Figure 1 is generated.

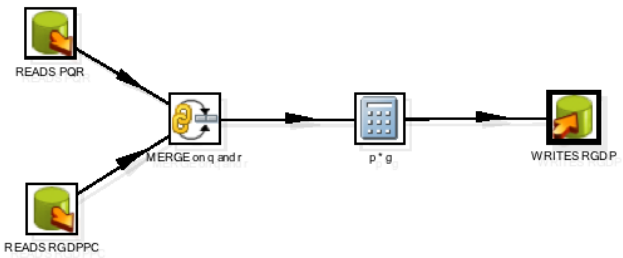

Figure 1: Example of schema mapping deployment as ETL flow

It is apparent that a complete execution of the data flow generates all the tuples implied by the tgd, since every tuple in the sources is fed into the stream and treated exactly once.

[14]http://kettle.pentaho.com/

6. EXLENGINE: ARCHITECTURAL OVERVIEW

EXLEngine is the software system used by Bank of Italy to support the execution of an EXL program. It is metadata-driven in the sense that the definitions of cubes (elementary or derived) and dependencies among them, expressed in terms of EXL statements, guide its runtime behavior. It refers to a data model called Matrix (for which we have described the cubes, omitting the illustration of the other aspects) to structure metadata definitions and to support the various features needed in the typical statistical data production process. A major example of this kind of features is historicity, that is, the time-dependence of cubes and programs, which is handled by the system. EXL statements are written by statisticians by means of IDE tools validating the programs both syntactically and semantically. System administrators trigger all the subsequent process, involving the translation and execution of the statements in the target systems.

Figure 2 represents a data view of the engine architecture and points out the data flows.

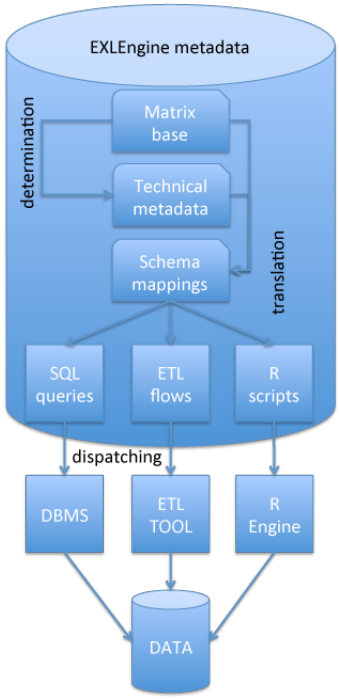

Figure 2: The architecture of EXLEngine

There is a *determination engine*, the component that has the responsibility to decide what cubes have to be calculated in the system. Indeed, in a production environment, EXLEngine handles a number of programs at the same time, which globally define a graph of dependencies among all the stored cubes. In this graph, which is actually a DAG (directed acyclic graph) for the aciclicity of EXL programs mentioned in Section 3, nodes represent the cubes, while there is a directed edge from a cube A to a cube C, if C is calculated from A in an EXL statement.

EXL programs need to be run, and derived cubes calculated,

when some values in elementary cubes (the leaves of the graph) change. Thus, the determination component detects what cubes have changed and performs a depth-first visit of the graph starting from the corresponding nodes, to produce a topologically sorted graph of all the cubes that need to be calculated according to the dependencies induced by EXL expressions. This means that it isolates a set of dependencies and dynamically builds the EXL program to be run.

The obtained graph is then partitioned by the determination engine into subgraphs. Each of them will be coherently delegated to a single target system; technical metadata are used to specify what specific target system is the most suitable to calculate each cube according to the specificity of the involved operators.

The *translation engine* for each set of cubes in a subgraph, considers the generating EXL expressions and performs the schema mapping generation algorithm producing an intermediate abstract representation of them. Then it translates the schema mappings into the executable translations according to the selected target systems.

All the activities described so far can be efficiently performed off line or at the startup of the system. In such a way the system decouples their computational time from the one of the actual statistical calculation. It follows that the metadata-driven approach of EXLEngine does not affect the global elapsed time for calculations.

In fact, at runtime, a *dispatcher* component assigns every subgraph to a specific target system, also applying parallelization and optimization patterns. Each *target engine* then only executes its native code and produces the results.

Real runs involve steps performed by several target engines, therefore they have to share the data they act on. Some engines, such as DBMSs, include storage capabilities and so can efficiently operate directly on data: a query dispatched to a DBMS will be directly executed on data stored in it.

By contrast, other engines can only calculate data but do not involve a storage system. In some situations, the dispatcher can provide them with the data they have to operate on and then handle the result storage. More often, there are meaningful performance advantages to let engines extract data from a storage system themselves. This implies that native scripts generated by the translation engine, are often engineered with data access primitives. Moreover, in the solution illustrated so far, every tgd is translated into a read-write statement in the target system. Actually, it is not necessary that all the intermediate steps are stored back into the system as intermediate cubes can be irrelevant. It follows that the whole approach can be easily reformulated in terms of creation of relational views, or intermediate R or Matlab structures for temporary cubes.

7. RELATED WORK

This paper aims at formalizing the link between statistical programs and their executable form as well as illustrating a working system. Statistical programs are implemented in several languages and with respect to different models. Here we make specific reference to the Matrix data model and to the EXL language [8, 9, 11], both developed by Bank of Italy, but any other language could have been considered: in fact, our approach is general since it does not directly link EXL to its executable form; by contrast, they are decoupled by means of schema mappings.

To the best of our knowledge, the link between any statistical language (and the respective reference model) and its implementation has never been pursued nor formalized.

This work aims at filling this gap by using the theoretical device of schema mappings. Schema mappings and their properties have been analyzed in detail in a number of works. Here we mostly leverage on Fagin et al. [13] definitions.

The theoretical process leading from statistical expressions to executable schema mappings is somehow twofold. First expressions are translated into schema mappings, then mappings are turned into their executable form. This approach leverages on the experience of Atzeni et al. [2, 4] on MIDST, a solution for model-independent schema and data translation; these works propose model-independent solutions based on the composition of elementary translations. Elementary translations are specified as Datalog programs acting on a pivot metamodel. Here we face the problem from a slight different perspective and do not foster a pivot metamodel; instead, we use mappings as a formal abstract language acting as a hub between an abstract transformation specification and its implementation.

The approach to the execution of the generated schema mappings delegates the effort to the target systems and does not involve a centralized execution. This is coherent with the runtime extensions [1] to the mentioned translation approaches, where translation metadata are determined in advance and only executable scripts are passed to the target systems. Approaches and algorithms for the execution of schema mappings in SQL systems can be also found in Clio [12, 15] and in +Spicy [18]. Similarly, in Orchid [10] a formal approach to the integration of schema mappings and ETL is presented. However, Orchid perspective is slightly different: it maps both ETL flows and schema mappings into a common model in order to integrate them. Here we abstract the specific representation of ETL flows and only consider schema mappings as a part of the system metadata.

8. CONCLUSIONS

The main contribution of this work is the formalization of the relationship between common tools used by statisticians (within a unifying language to write conceptual programs) and a translation approach generating their executable version. The formal device of schema mappings is adopted as an intermediate step allowing for a uniform translation of programs into their running form in the target systems.

To the best of our knowledge this is the first proposal in this direction.

The approach shows that a statistical program, written in an abstract specification language, can be correctly translated into a schema mapping: in particular it turns out that the solution to such a data exchange setting coincides with the outcome of the statistical program. Concretely, we

presented EXLEngine, the engineered software architecture adopted in Bank of Italy implementing the described algorithms.

The approach still has some limitations that are being addressed. Many operators are particularly easy to be supported on certain target systems while there are issues on others. Moreover, a major challenge is devising further translations of schema mappings in order to widen the spectrum of supported target systems.

9. REFERENCES

[1] P. Atzeni, L. Bellomarini, F. Bugiotti, F. Celli, and G. Gianforme. A runtime approach to model-generic translation of schema and data. In *Inf. Syst.*, pages 269–287, 2012.

[2] P. Atzeni, L. Bellomarini, F. Bugiotti, and G. Gianforme. MISM: A platform for model-independent solutions to model management problems. In *J. Data Semantics*, pages 133–161, 2009.

[3] P. Atzeni, P. Cappellari, and P. A. Bernstein. Modelgen: Model independent schema translation. In *ICDE Conference*, pages 1111–1112. IEEE Computer Society, 2005.

[4] P. Atzeni, P. Cappellari, R. Torlone, P. A. Bernstein, and G. Gianforme. Model-independent schema translation. *VLDB J.*, 17(6):1347–1370, 2008.

[5] C. Beeri and M. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[6] P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD Conference*, pages 1–12. ACM, 2007.

[7] P. J. Brockwell and R. A. Davis. *Introduction to Time Series and Forecasting*. Springer, 2002.

[8] V. Del Vecchio. Statistical data and concepts representation. *Bank of Italy*, 1997. `http://www.bancaditalia.it/statistiche/quadro_norma_metodo/modell_SIS/StatisticalDataAndConceptsRepresentation.pdf`.

[9] V. Del Vecchio, F. Di Giovanni, and S. Pambianco. The "matrix" model - unified model for statistical data representation and processing. *Bank of Italy*, 2007. `http://www.bancaditalia.it/statistiche/quadro_norma_metodo/modell_SIS/matrixmod.pdf`.

[10] S. Dessloch, M. Hernández, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: Integrating schema mapping and etl. In *ICDE*, pages 1307–1316, 2008.

[11] F. Di Giovanni and D. Piazza. Processing and managing statistical data: a national central bank experience. *Bank of Italy*, 2009. `http://www.czso.cz/conference2009/proceedings/data/process/piazza_paper.pdf`.

[12] R. Fagin, L. Haas, M. Hernández, R. Miller, L. Popa, and Y. Velegrakis. Clio: Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications*, pages 198–236, 2009.

[13] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[14] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[15] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD Conference*, pages 805–810. ACM, 2005.

[16] P. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75, 2005.

[17] E. Malmborg and B. Sundgren. Integration of statistical information systems - theory and practice. In J. C. French and H. Hinterberger, editors, *SSDBM*, pages 80–89. IEEE Computer Society, 1994.

[18] G. Mecca, P. Papotti, and S. Raunich. Core schema mappings: Scalable core computations in data exchange. *Inf. Syst.*, 37(7):677–711, 2012.

[19] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *VLDB*, pages 264–277. Morgan Kaufmann, 1990.

[20] J. O. Ramsay, G. Hooker, and S. Graves. *Functional Data Analysis with R and Matlab*. Springer, 2009.