Noname manuscript No.
(will be inserted by the editor)

Executable Schema Mappings for Statistical Data Processing

**Paolo Atzeni · Luigi Bellomarini ·
Francesca Bugiotti · Marco De Leonardis**

Received: date / Accepted: date

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. We illustrate here a proposal where a high-level language, EXL, is used for the declarative specification of statistical programs, and a translation into executable form in various target systems is available. The language is based on the theory of schema mappings, in particular those defined by a specific class of tgds, which we actually use to optimize user programs and facilitate the translation towards several target systems. The characteristics of such class guarantee good tractability properties and the applicability in Big Data settings. A concrete implementation, EXLEngine, has been carried out and is currently used at the Bank of Italy.

Keywords Schema mappings; statistical data; scalable data processing; ETL

1 Introduction

The generation of statistical products, in the form of finished and deliverable data artifacts, is the primary goal of statistical information systems. The products can be directly addressed to the public, as it happens for publications in

P. Atzeni
Università Roma Tre

L. Bellomarini
Università Roma Tre & Bank of Italy

F. Bugiotti
CentraleSupélec & LRI

M. De Leonardis
Hewlett-Packard

official statistics, or be specifically designed to be used in research, analysis, and decision making. In all cases, data processing is of primary importance in the production process and consists of several integrated human and automatic data manipulation activities that transform the raw data, coming in various formats from different sources, into the statistical products.

With the recent progress in computing tools and techniques, many activities in statistical information systems can be effectively automated, especially in the systematic production of official statistics, where a definite and precise set of transformation steps can be specified. A major, long term goal in this respect would be the possibility to write high-level statistical programs in a language oriented to the actual business of statistics and independent of any specific technology. Such programs should also be easy to execute in real systems and exhibit good scalability. Indeed, similar problems arise in many areas of software development and especially in the database field, where a lot of consideration has been devoted to the approaches that try to provide support to transformations, by means of methods and languages for the declarative specification of actions for the extraction and manipulation of data.

A major contribution in this direction comes from logic-based frameworks, which provide formal devices and languages to describe transformations, perform reasoning, query answering and data exchange tasks [3,15,32]. These frameworks try to balance the richness of modeling features and the complexity of performing the mentioned tasks, with the ultimate goal of achieving high expressive power and suitability for Big Data settings [9].

Among them, schema mappings can be considered as a very encompassing framework [11]. It is also worth mentioning that while most of the theory of mappings has been developed for the relational model, arguments for more general, model-independent approaches have been made [7]. They also have proven applicability: for example, the close relationship between schema mappings and ETL (Extract - Transform - Load) executable flows, typically used in data warehousing, has been practically clarified by many works, e.g., the Orchid prototype [23], and had been understood since Clio early studies [31].

The need for automatic translation of high-level statistical programs into various executable forms is a central problem, because of the presence of several kinds of execution engines (the target systems) used in practice. The most adopted general-purpose engines are relational database systems, since they guarantee a simple and solid SQL interface, as well as a resilient architecture and high performance and scalability for small and local operations. Many other engines are domain dependent, usually third-party tools specifically designed to address the needs of statistical calculations: examples are R, Matlab, STATA, and eViews. The languages used in those engines are also domain dependent and require advanced programming skills to build efficient applications even for simple tasks. Furthermore, the proliferation of different languages results in difficulties in the reuse of code. For these reasons, the need for a unifying approach, linking the specification of high-level statistical programs and the underlying IT solutions with scalability guarantees has clearly emerged [24].

In this paper we leverage the effectiveness of schema mappings and the huge expertise on them in the database community, to build a theoretical and practical bridge between a high-level specification of a statistical program and its execution in many different systems (relational databases, statistical engines, ETL tools).

Our conceptual goal is to formalize the relationship between programs expressed at high level, in terms of statistical concepts, and their executable form in the systems. For this, we use schema mappings as an intermediate representation between the statistical concepts and the languages of target systems, where real execution takes place. Our schema mappings are based on a class of tgds (tuple-generating dependencies), which for our purposes have a good balance between expressive power and tractability. Moreover, we provide a system-independent algorithm that converts the mappings into the various execution languages. We show that the result of the execution of the schema mappings is equivalent to the procedural application of the programs. To achieve this target, we formulate a data exchange problem instance out of the generated schema mappings and show that its solution is equivalent to the effect of the statistical program. As a consequence of the tractability of the transformation tasks in our class of tgds, we also argue for the scalability of our approach.

Our practical goal is decoupling the abstract specification of statistical procedures from their technical implementation, while guaranteeing the generation of efficient executable code. Indeed, we use schema mappings for multi-query optimization, that is, the combination of different program directives into fewer queries, to favor efficient execution in the target systems. To this end, we introduce a novel notion of composition of mapping dependencies in the presence of mathematical operators; we characterize the general problem of composability of dependencies by means of a necessary and sufficient condition, and show how we practically leverage it in our algorithms to generate optimized executable mappings.

This paper is based on the experience of two of the authors at the Bank of Italy, which has shared these objectives, playing for decades a key role in the process of standardization and devising Matrix [22], a statistical data model, and a high-level language named EXL [22] (EXpression Language) used to write statistical programs over cubes (involving sum, difference, aggregations of cubes etc.) exploiting the potential of the Matrix model (at the specification level, with limited executability).

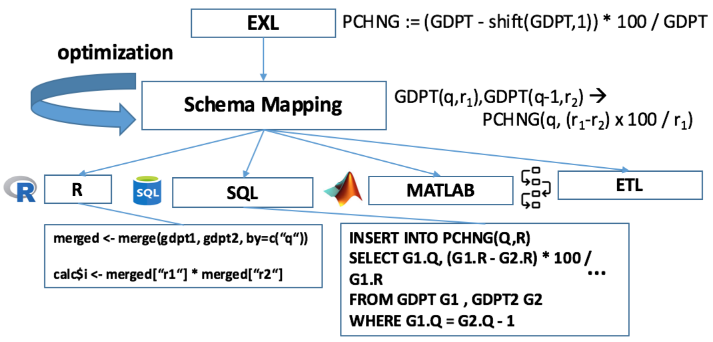
Here, we present an executable system, *EXLEngine*, in which EXL is used for high-level specifications. The programs specified by Bank statisticians are fed into EXLEngine, which translates them into schema mappings. The schema mappings are optimized with techniques based on the theory of mapping composition and finally translated into an executable form for the specific target system. For example, mappings (and so the original program) could be translated (i) into SQL to delegate the execution to a DBMS; or (ii) into ETL jobs to pass the execution to a specialized stream-like architecture such as an ETL engine; or (iii) into a specialized language for a statistical tool (such as R, Matlab, etc.). The system is actually used at the Bank of Italy for the calculation of national financial statistics.

A preliminary version of the system has been presented in [4]. Here, we extend such a paper, with the following main addictions: we formalize and develop the role of schema mappings in the framework, giving a formal characterization of the results and propose novel optimization techniques based on composition, in mappings with arithmetic operations and aggregations. Moreover, we add many details.

The remainder of the paper is organized as follows. In the next Section we give an overview of our approach. In Section 3, we recall some preliminary notions about schema mappings. In Section 4 we present EXL, a specification language for statistical programs. In Section 5 we illustrate how to generate schema mappings out of statistical programs and focus on how we compose them to favor optimization. We address correctness and scalability by studying a data exchange setting formulated with the generated mappings. We study and characterize the general problem of composing dependencies in presence of mathematical operators. In Section 6 we show how to translate the schema mappings into various executable forms, such as SQL queries or ETL jobs. The implementation of EXLEngine is then briefly illustrated in Section 7 along with preliminary but meaningful performance evaluations. In Section 8 we discuss related work and finally in Section 9 we draw our conclusions.

2 Overview

A conceptual representation of our approach is sketched in Figure 1.


EXL
PCHNG := (GDPT - shift(GDPT,1)) * 100 / GDPT
optimization
Schema Mapping
$GDPT(q,r_1),GDPT(q\text{-}1,r_2) \rightarrow PCHNG(q, (r_1\text{-}r_2) \times 100 / r_1)$
R
SQL
MATLAB
ETL
...

```
merged <- merge(gdpt1, gdpt2, by=c("q"))

calc$i <- merged["r1"] * merged["r2"]
```

```
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
```

Fig. 1: The complete execution process: from EXL to the executable languages.

1. EXL programs are translated into schema mappings; 2. schema mappings are logically optimized; 3. schema mappings are translated into a form that is executable in the target systems.

Let us now go through the process by using an example, a small statistical program, for which we show the specification in EXL in Figure 2 and a sample of the respective values in Figure 3. It 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; the program finally calculates an indicator over GDP called FGDP. In EXL, uppercase strings denote relations. All of them are assumed to have a key constraint on one or more attributes (named *dimensions*) and carry a numeric value in a specific attribute (the *measure*). Notice that unlike usual logic formalisms, the syntax omits any indication of the attributes for compactness of representation.

```
PQR := avg(PDR, group by quarter(d), region)
RGDP := RGDPPC * PQR
GDP := sum(RGDP, group by quarter)
GDPT := stl_T(GDP)
PCHNG := (GDPT - shift(GDPT,1)) * 100 / GDPT
PM := PM2 * 0.2
FGDP := GDP * PM * PCHNG/100
```

Fig. 2: An EXL program related to the national GDP.

PDR		
Day	**Reg**	Pop
1	A	100
	B	200
2	A	105
	B	202
...	...	...
91	A	120
	B	210
92	A	130
	B	240
...	...	...
182	A	125
	B	215
183	A	135
	B	245
...	...	...
274	A	140
	B	250
275	A	160
	B	270
...	...	...

PQR		
Quart	**Reg**	Pop
1	A	102.5
	B	201
2	A	125
	B	225
3	A	130
	B	230
4	A	150
	B	260

RGDPPC		
Quart	**Reg**	Gdppc
1	A	1000
	B	2000
2	A	1200
	B	2200
3	A	1100
	B	2150
4	A	1120
	B	2150

RGDP		
Quart	**Reg**	Gdp
1	A	102500
	B	402000
2	A	150000
	B	495000
3	A	143000
	B	494500
4	A	168000
	B	559000

GDP	
Quart	Gdp
1	504500
2	645000
3	637500
4	727000

GDPT	
Quart	Gdp
1	504000
2	645000
3	637000
4	720000

PCHNG	
Quart	Change
2	21.8
3	-1.25
4	11.52

PM2	
Quart	Factor
2	4
3	2
4	6

PM	
Quart	Factor
2	0.8
3	0.4
4	1.2

FGDP	
Quart	Fgdp
2	112488
3	-3187
4	100500

Fig. 3: Sample values for the program in Figure 2.

In the program in Figure 2, PDR(d, r, p) represents the population p of a region r on day d. PQR(q, r, p) represents the same population p 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 by computing the average of the daily values (by means the `avg` function). RGDP(q, r, g) is the regional gross product g in a quarter, and it is obtained (line 2) by multiplying RGDPPC (regional gross domestic product per capita) of the quarter and the (previously calculated) average population in the same quarter. GDP(q, g) is then obtained (line 3) as the sum of RGDP(q, r, g) over regions. Then, in line 4, the *seasonal decomposition*[1] operator stl_T is used to isolate the trend GDPT(q, g). The PCHNG(q, c) is obtained (line 5) as the difference between the values of GDPT in two consecutive quarters, divided by the trend itself and multiplied by 100 (to have a percentage).[2] Finally (line 7) we also calculate an indicator FGDP(q, f) by multiplying GDP by PCHNG, the constant 100, and a forecast model PM(q, f), in turn obtained by scaling another model PM2(q, f) (line 6).

Our program is first translated into schema mappings and then into an executable form for the back-ends. In Figure 1, we show such translations for statement (5) of the program in Figure 2, first into tgds and then into executable forms (e.g., SQL and R).

In Section 5, we discuss the correspondence between EXL (presented in Section 4) and our class of tgds.[3] Indeed, the tgds we need are extensions of those commonly used in data exchange settings, because we also have tuple-level operators and aggregations. These schema mappings represent in our approach an intermediate system-independent step, which then needs to be translated into an executable form. However, a direct one-to-one translation of the mappings would not exploit the target system optimizer at best. To this purpose, in the full technique we present in Section 5, we propose and apply an optimization algorithm that combines all the pairs of dependencies that can be conveniently composed into a single one.

In general, for the mappings, many translations are indeed possible and we support different target systems. Beside relational databases, there are other possible target systems and languages, which include those specifically designed for statistical elaborations, such as R and Matlab. They are typically vector or matrix oriented and offer a number of powerful statistical functions. The translation from a tgd into such target languages is often very natural.

ETL platforms are another interesting kind of target system that we support, as as we will discuss in Section 6. A statistical program can be intuitively seen as an ETL job composed of a number of flows each representing a tgd statement. In this context 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

[1] 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 "variations", ignoring seasonal, cyclic (and stochastic) ones [14,37].

[2] Note that for the first quarter the PCHNG is not meaningful.

[3] As we will see in Section 5, we also have some egds, which enforce the functional nature of EXL relations.

or user-defined algebraic or statistical calculations; output steps, writing the results back into the system. For each relation in the left-hand side (lhs) of the tgd 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.

As we said, the approach we propose in this paper (and just sketched above) has been implemented in the EXLEngine system, used at the Bank of Italy for the generation and execution of schema mappings out of declarative statistical programs.

3 Foundations of Schema Mappings

Before introducing the EXL language and explaining our approach to the generation of executable mappings, let us provide some formal background.

A *relation schema*, denoted by a *relation name* R, is a finite collection of *attributes*, $R(A_1, \ldots, A_k)$. A *database schema* (or, simply, a *schema*) is a finite collection of relation schemas, $\mathbf{R} = \{R_1, \ldots, R_n\}$, with distinct relation names. An *instance* over a relation schema $R(A_1, \ldots, A_k)$ is a finite set of *tuples*, over the schema of R, each of the form $(v_1, \ldots, v_k)$, where each A_i is an attribute of R and each v_i is its corresponding *value*. An instance over a schema $\mathbf{R}$ is a finite set of *facts*, each of the form $R(v_1, \ldots, v_k)$, where $R(A_1, \ldots, A_k)$ is a relation schema in R and $(v_1, \ldots, v_k)$ is a tuple over R. In the common practice, the values v_i are taken from two (infinite) sets, a set $\mathcal{C}$ of *constants* (attribute values), and a set $\mathcal{V}$ of *variables* (also called *labeled nulls*). Variables are used to handle transformations where specific constants cannot be found: ideally, relations contain values, but sometimes they are not known.

Mappings: Given two database schemas $\mathbf{S}$ (source) and $\mathbf{T}$ (target), a *schema mapping* (or just *mapping*) M is a binary relation over all the possible instances of the two schemas, that is, $M \subseteq \mathbf{S} \times \mathbf{T}$. In mappings, instances of $\mathbf{S}$ are assumed to take values from the domain $\mathcal{C}$, while instances in $\mathbf{T}$ take values from $\mathcal{C} \cup \mathcal{V}$; also, $\mathbf{S}$ and $\mathbf{T}$ do not have relation names in common.

Data exchange: Given an instance $I \in \mathbf{S}$, we say that $J \in \mathbf{T}$ is a *data exchange solution* (or simply *solution*) for I under a mapping M, if $(I, J) \in M$. As mappings are relations, in general there can be multiple solutions $J \in \mathbf{T}$ for a given instance $I \in \mathbf{S}$.

Dependencies: If $\mathbf{S}$ and $\mathbf{T}$ are schemas with no common relation names, a sentence σ is a *source-to-target tuple-generating-dependency* (st-tgd) from $\mathbf{S}$ to $\mathbf{T}$ if it has the form: $\forall \mathbf{x}(\phi(\mathbf{x}) \rightarrow \exists \mathbf{y} \psi(\mathbf{x}, \mathbf{y}))$, where $\mathbf{x}$ and $\mathbf{y}$ are two disjoint sets of variables of $\mathcal{V}$, $\phi(\mathbf{x})$ (the left-hand-side, abbreviated as LHS) is a conjunction of atoms with relation names (*relational atoms* in the following) over $\mathbf{S}$, with variables in $\mathcal{V}$ and constants in $\mathcal{C}$, and $\psi(\mathbf{x}, \mathbf{y})$ (the right-hand-side, RHS) is a conjunction of atoms with relation names over $\mathbf{T}$, built with variables in $\mathcal{V}$ and constants in $\mathcal{C}$. A sentence σ is a *target tuple-generating-dependency* (t-tgd), if it has the same form as st-tgds, but $\phi(\mathbf{x})$, the lhs, is a conjunction of relational atoms over $\mathbf{T}$. Given target schema $\mathbf{T}$, a sentence σ is an *equality-generating-*

dependency (egd) over $\mathbf{T}$, if it has the form: $\forall \mathbf{x}(\phi(\mathbf{x}) \rightarrow x_i = x_j)$, where x_i and x_j are variables of ϕ.

Specification of mappings: Let Σ be a set of dependencies. We say that a mapping M from $\mathbf{S}$ to $\mathbf{T}$ is specified by Σ (and we write $M = (\mathbf{S}, \mathbf{T}, \Sigma)$) if, for every $(I, J) \in \mathbf{S} \times \mathbf{T}$, it is the case that $(I, J) \in M$ if and only if (I, J) satisfies Σ. We say that $\mathbf{S}$ and $\mathbf{T}$ are the source and target schemas of M, respectively.

As we will see, in our setting, schema mappings are specified through a set of st-tgds, *non-recursive* t-tgds extended with operators, and egds.

Chase: The solutions to a data exchange setting with st-tgds, t-tgds and egds are usually obtained by means of the *chase* procedure [26,32]. Intuitively, it "applies" constraints to an instance, "forcing" their satisfaction. For tgds, this means adding, if needed, new tuples corresponding to the rhs for each set of tuples that unify with the lhs. New values can be "invented" with various approaches: a common one is *Skolem functions*, which deterministically produce new values for the target given n-uples of source values. Instead, forcing egds satisfaction means unifying values, assigning variables or equating constants, which may lead to failure. 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 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 general, the chase is not guaranteed to terminate, due to the possible presence of recursion in t-tgds with infinite creation of new labeled nulls. However, if t-tgds are *weakly acyclic* (i.e., the procedure cannot enter a cycle where the application of a t-tgd creates a labeled null for each pass in the cycle), then the chase is guaranteed to terminate and return a solution, if one exists, or fail, if none exists (an egd is violated). If there is one, the returned solution is known as *universal* and is unique up to homomorphism [38].

4 The Expression Language

In this Section we illustrate the language we adopt for the high-level specification of statistical programs. Indeed, we present a formalization of the language actually used at the Bank of Italy in these activities.

Section 4.1 gives an overview of the language. Then, Sections 4.2 and 4.3 give details about two kinds of operators that are present in the language: *tuple-level* and *multi-tuple*, respectively. Numeric examples of application of the operators can be found in Figure 3.

4.1 The language

Let us now turn our attention to *EXL (EXpression Language)*, a specification language for statistical programs over relations, and formalize the language used by the Bank of Italy in this context. We have already seen an example in Section 2. The language handles relations that model tabular functions. In particular, they are required to have a key constraint, defined on a subset of their attributes (the *dimensions*, which are the independent variables in the tabular function) and have exactly one numeric attribute (the *measure*, which is the dependent variable of the tabular function).

The manipulations on relations are specified in the form of input/output transformations, encoded by *operators*, which take relations and other parameters as input and return relations. The operator-based formalism is particularly suitable for an easy specification of high-level programs, with an approach that is similar to that of ETL flows.

More formally, an EXL statistical program is a sequence of *statements.* A statement is an assignment (denoted by the symbol :=) where the left-hand side (lhs) is a relation identifier and the right-hand side (rhs) is an *expression.*

In an EXL program, relation 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 expression in a statement (which specifies the definition of a derived relation) contains only elementary relations or relations that are derived in previous statements of the program. So, no recursive definition of derived relations is allowed, as this is not needed in the statistical applications of interest. Moreover, as a relation defines a function, there needs to be a unique way to obtain it, and so a relation 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 relation are calculated. They can be recursively defined as follows:

– a relation identifier (e.g. R) is an expression; we use the term *relation literal* for this base case; the type of the expression is the same as that of the relation;
– 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 as well as various 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.

The common feature of operators is that, obviously, an operator produces a result relation (a function with at most one value for each dimensions tuple) from one or more input relations. In general, the value of the relation on a dimensions tuple may depend on the values of several input 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 (typically 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 parentheses) 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 $\log(2, x * 3)$, or the shift in the time series we already saw in the example in Section 2. Structural elements include, for instance, the specification of the grouping dimensions in aggregation operators, as we will show in Section 4.3.

4.2 Tuple-level operators

Let us now concentrate on tuple-level operators, which are the ones expressing tuple-to-tuple correspondences. They can be unary or n-ary.

Unary operators have one single operand relation and possibly some extra scalar parameters. This category includes the most natural operators on the measures of relations (such as sum, subtraction, product, division by a constant, increment, logarithm, exponential, trigonometric function). Here, the resulting relation has the same dimensions as the operand and contains a tuple for each tuple of the operand, with the same values for the dimensions and a calculated one for the measures: for example, given the statement $K := 1+E$, we have that each tuple of K is defined after each tuple of E with the same values for the dimensions and the measure calculated as 1 plus the value of the measure of E, tuple-wise.

Other unary operators operate on the dimensions. The most common here is the *shift*, which is essentially a sum on the values of a numeric or time dimension. The semantics of the time shift with parameter s is that for each value t on which the operand relation is defined, the result relation is defined on dimension values $t + s$ and with the same value: given expression e, we have that $\text{shift}(e(t + s)) = e(t)$, for all t.

N-ary operators have two (or even more, but this is not essential here) operand relations, generating a third one as a result. The two operands and the result as well have the same dimensions (same name and type for each), and the semantics is defined as expected, for each dimensions tuple. A nontrivial issue is how to deal with relations that have the same dimensions but their values exist on different dimensions tuples: here different versions exist, we mainly refer the simplest, which produces the result tuple only for dimensions tuples that appear in both relations, but there are others assuming a default value for the "missing" tuples (for example, in the sum operator, we could have zero as the default value). Indeed the language supports also a more general case that operates on relations with different but compatible dimensions: there is at least one operand that comprises the dimensions of each of the others (and all of those dimensions appear in the result).

4.3 Multi-tuple operators

The second class, that of multi-tuple operators, includes many of them, which are indeed important in the production of statistical data, as they restructure relations, calculating new values from sets of previous ones. Among them, we have a significant subclass whose elements can be considered as black boxes, because they receive one relation in input and transform it by producing another relation. 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" relations, by applying a specific arithmetic operator (for example sum, max, min, or average) to the values of the relation 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 them (for example, the application of the `quarter` function to a date dimension, as we saw in statement (1) in the example in Section 2). The semantics is essentially the same as we have in SQL aggregate queries: the result relation 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[4] of values in relation e that are associated with dimensions tuples (in e) that coincide with $(x_1, \ldots, x_k)$ on *dimensionList*. Then, the value of the result relation on $(x_1, \ldots, x_k)$ is the result of applying function *aggr* to the bag V. The relation tuple exists only if the bag V is non-empty.

[4] That is, repeated elements are meaningful.

5 Generating Schema Mappings from Statistical Programs

In this section we present our various results on schema mappings in our setting. In Section 5.1 we show how schema mappings can be generated out of an EXL statistical program. In Section 5.2 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. We discuss the complexity and the scalability of the approach in Section 5.3. Then in Section 5.4 we show how we combine the generated mappings to allow multi-query optimization. In Section 5.5 we characterize the general problem of composability and present useful result that effectively apply to our context. Finally, in Section 5.6, we present our optimization technique.

5.1 The generation of schema mappings

With respect to the theory of schema mappings recalled in Section 3, we need to make some extensions to the language for dependencies. In fact, as we saw in Section 4, 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.

The source relational schema $\mathbf{S}$ contains a relation $F_i(X_{i,1}, \ldots, X_{i,n_i}, Y_i)$ for each relation in the EXL program of interest. The target schema $\mathbf{T}$ contains the same relations, which we however need to rename, since we assume (as usual [27]) $\mathbf{S} \cap \mathbf{T} = \emptyset$.

So, for each $F_{S,i}(X_{i,1}, \ldots, X_{i,n_i}, Y_i) \in S$, we have a relation $F_{T,i}(X_{i,1}, \ldots, X_{i,n_i}, Y_i) \in T$, and a st-tgd in Σ 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 tgds 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 to enforce the key constraints on dimensions. This is modeled by means of egds of the form:

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

Finally, we have t-tgds that correspond to the EXL statements. We start, in the following discussion, from a scenario where each expression is decomposed into many expressions, each comprising only one operator.[5] We argue that:

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

first, this assumption does not cause loss of generality, as any statement can be rebuilt with simple substitutions (and indeed the corresponding mappings will be composed, as we will show in Section 5.4); second, working with basic statements maximizes the possibility to reuse intermediate partial results (which also will be more clear in the following). For example, statement (5) in the example in Section 2 is decomposed into four statements, as follows:

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

According to these hypotheses, all EXL statements have the form

– R `:=` op `(`R_1`, ...,` R_k`)`

where the operator *op* might have the syntax with special symbols (for example the infix syntax with `+` or `*`) 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 relation is computed value by value, by applying a function to the individual values of the input relations 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 relation 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, respectively)

– `R2 := 3 * R1`
– `R5 := R3 + R4`
– `R7 := shift(R6,1)`

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

– $R_1(x_1, x_2, y) \to R_2(x_1, x_2, 3 \times y)$
– $R_3(x_1, x_2, y_1), R_4(x_1, x_2, y_2) \to R_5(x_1, x_2, y_1 + y_2)$
– $R_6(t, y) \to R_7(t - 1, y)$

These tgds are indeed a bit more complex than those usually found in data exchange settings, but their semantics 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, in the values that are created by these tgds are uniquely defined by operators, therefore the generated tuples are uniquely defined as well 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 relation.[6] 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 relation could even depend on all the tuples of the input relation.

Here, given a statement of the form

– `R2 := aggr ( R1 , group by D1, ..., Dk )`

assuming that $D_1, ..., D_k$ are the first k dimensions of R_1, we would have the following tgd:

– $R_1(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n, y) \rightarrow R_2(x_1, \ldots, x_k, aggr(y))$

whose semantics would be:

– for every different tuple $x_1, \ldots, x_k$ in the projection of R_1 on $D_1, ..., D_k$, there is a tuple $x_1, \ldots, x_k, y'$ in R_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 bag of values that the measure has in the tuples of R_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 [36], where the basic idea 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 over EXL statements.

We report below a set of t-tgds, derived from the EXL statements of our running example according to the procedure we have just described. In statement (1) we have a scalar function (quarter) that operates on the values of a dimension together with the average function to allow the roll up, and we have it in the same way in the tgd. In statements (5) and (7) we have four and three operators respectively and the resulting tgd takes care of all of them (altogether), as we will explain in Section 5.4.

(1) $\mathrm{PDR}(q, r, p) \rightarrow \mathrm{PQR}(\mathrm{quarter}(q), r, \mathrm{avg}(p))$
(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) $\mathrm{GDPT}(q, r_1) \wedge \mathrm{GDPT}(q-1, r_2) \rightarrow \mathrm{PCHNG}(q, (r_1 - r_2) \times 100/r_1)$
(6) $\mathrm{PM2}(q, r) \rightarrow \mathrm{PM}(q, r * 0.2)$
(7) $\mathrm{GDP}(q, r_1) \wedge \mathrm{PM}(q, r_2) \wedge \mathrm{PCHNG}(q, r_3) \rightarrow \mathrm{FGDP}(q, r_1 \times r_2 \times r_3/100)$

[6] The case with several relations is indeed possible and we will discuss it in Section 5.4.

5.2 Correctness of the schema mappings

Let us now argue for the correctness of the schema mappings generated out of the EXL programs in the way we illustrated in Section 5.1. To this purpose, we consider the data exchange problem associated with the EXL program. Given that an EXL program always terminates (due to the acyclicity of the statements), we prove that the data exchange problem always has a solution, which can be found by means of a suitable variation of the chase (Th. 1).

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 (Th. 2).

Theorem 1 *Given a mapping $M = (\mathbf{S}, \mathbf{T}, \Sigma)$ and an instance I of* $\mathbf{S}$*, where* $\mathbf{S}$ *is the relational schema containing the input relations in the EXL program,* Σ *is a set of st-tgds, non-recursive t-tgds with operators and egds (the dependencies are assumed to be built as described in Section 5.1), and* $\mathbf{T}$ *is the relational schema containing the output relations in the EXL program, the chase terminates, succeeds and solves the data exchange problem.*

Proof The data exchange problem we have in our case exhibits some differences with respect to the usual case (Section 3), due to the presence of operators.

Independently of their specific behaviors, tuple-level and multi-tuple operators produce new values in the target. We argue that both categories can be seen as pre-defined Skolem functions.

Tuple-level operators are functions that deterministically generate a new value in a target tuple given a combination of values in a source tuple, therefore they are Skolem functions by definition. For multi-tuple operators, we adopt a stratified semantics, in which the application of the tgds in the chase is constrained. Rather than allowing the applications of tgds in any order, we follow a stratified approach: we consider the total order induced by the dependency graph, calculated from the EXL program and apply tgds in that order.[7] In this way, when we apply aggregations, the values of a tuple in the target is calculated on the basis of the complete bag of values in the source, which implies that repeated applications will produce the same value. This makes multi-tuple operators also suitable as pre-defined Skolem functions for the existential quantification.

Since the t-tgds adopted in our setting are non-recursive and so by definition weakly acyclic, the chase is guaranteed to terminate (see Section 3). Let us now consider egds. In general, egds can cause a failure of the chase, but in our setting this cannot happen since: 1. operators are well-defined and have a deterministic behavior (they produce the same output values for repeated input values); 2. each relation appears in the rhs of exactly one t-tgd, which prevents any conflicts in the target.

[7] 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.

This guarantees that in our setting the chase always terminates, returns a solution and solves the data exchange problem. □

Theorem 2 *Given a mapping $M = (\mathbf{S}, \mathbf{T}, \Sigma)$ and an instance I of $\mathbf{S}$, where $\mathbf{S}$ is the relational schema containing the input the relations in the EXL program, Σ is a set of st-tgds, non-recursive t-tgds with operators and egds (the dependencies represent are assumed to be built as described in Section 5.1), and $\mathbf{T}$ is the relational schema containing the output relation in the EXL program, the data exchange solution J is equivalent to the output of the EXL program.*

Proof (Sketch) 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 relation tuple generated by the EXL program and vice versa. We can claim that this is the case because of the way the tgds have been defined: for each statement we have a mapping with one tgd that generates one tuple (and exactly one) for each tuple generated by the corresponding statement. This can be proven, in tedious but straightforward way, for each of the operators, in both classes, tuple-level and multi-tuple. □

5.3 Scalability of the approach

Given the absence of recursion, the t-tgds of our data exchange setting are by definition in the weakly acyclic class, for which the data exchange problem is known to be polynomial in data complexity [33]. Although polynomial time data complexity is desirable for conventional settings, it can become unaffordable for Big Data applications, usual in the statistical context. In fact, even linear time data complexity can be considered prohibitive. In this respect, we observe that in spite of the expressive limitation posed by the complete absence of recursion (e.g., impossibility to express a transitive closure), our data exchange setting is in the highly tractable NLOGSPACE class. This can be easily proven on the basis of the tree structure of the chase, along the lines of similar results in reasoning and query answering in description logics [9, 16]. Although our approach is translation-based and so we rely on the features of the target optimizer, NLOGSPACE complexity guarantees that any EXL program can be implemented in a highly parallel way, or in other terms, that, when available, the parallelization features of the optimizers will be leveraged.

5.4 Composing the mappings

We have worked so far with elementary statements, each of which uses one operator, which is translated into a single, simple t-tgd (*elementary tgd*). As we will see in Section 6, each tgd can be translated into an executable program in the target system. Though intuitive and straightforward, this approach may lead to a target program composed of many simple steps (one operation each) and so unable to exploit the optimization capabilities of the specific target system. Indeed, each of these systems has its own optimization strategies, which can be applied to complex expressions and not to sequences of simple operations. In order to delegate the optimization responsibility to the target systems, we need to compact the statements passed to them, and we propose to proceed by composing the tgds in our mappings.

Let us start by introducing a suitable notion of *composition* for tgds. Let us first say that two tgds[8] σ_1 and σ_2 are *consecutive* if the atom in the rhs of σ_1 appears among the atoms in the lhs of σ_2. With reference to our language of tgds with operators, the general form of two consecutive tgds is the following:

$$\begin{aligned} \sigma_1 : \phi_1(\cdot, y_1, \ldots, y_n) &\rightarrow K_1(\cdot, \mathrm{op}_1(y_1, \ldots, y_n)) \\ \sigma_2 : \phi_2(\cdot, z_1, \ldots, z_m) \wedge K_1(\cdot, z) &\rightarrow K_2(\cdot, \mathrm{op}_2(z_1, \ldots, z_m, z)) \end{aligned} \tag{1}$$

In the above dependencies, ϕ_1 and ϕ_2 are conjunctions of relational atoms; the dot $(\cdot)$ notation represents any vector of variables that bind to the dimensions; variables $y_1, \ldots, y_n$ and $z_1, \ldots, z_m$ represent the measures of the conjuncted atoms in ϕ_1 and ϕ_2 respectively (one measure for each atom). K_1 in σ_1 is a relational atom, with any tuple of dimensions (which need not coincide with those of ϕ_1 as there may be aggregations), and one single measure calculated with the application of an operator op_1 to all the measures of ϕ_1. In atom K_2 the measure is calculated as the application of an operator op_2 to the measures of ϕ_2 and K_1. In σ_2, K_1 is the same relational atom as in σ_1, with the measure bound to variable z.

In the following, we will consider in general each operator op as the composition $\mathrm{op}^M \circ \mathrm{op}^T$ of an *n-ary tuple-level constituent* (op^T), applied first, and a *unary multi-tuple constituent* (op^M) with aggregations and grouping functions, applied on the result of the former. According to the general definition of composition of functions, for each tuple $x_1, \ldots, x_n$, we therefore have:

$$\mathrm{op}(x_1, \ldots, x_n) = (\mathrm{op}^M \circ \mathrm{op}^T)(x_1, \ldots, x_n) = \mathrm{op}^M(\mathrm{op}^T(x_1, \ldots, x_n))$$

As we will see, this formalization turns out to be very useful to guarantee the closure of our definitions in the recursive cases, where the composition of tgds is applied to the result of a previous one, possibly giving rise to operators with a hybrid (tuple-level and multi-tuple) nature. However, in the base cases of our settings, operators are elementary, that is, they are either tuple-level (hence only having the op^T constituent) or multi-tuple (hence only having the op^M constituent). In particular, when op_1 is simply a multi-tuple operator, ϕ_1 consists of one atom only, thus in the rhs of σ_1 we have only measure y_1 (for that atom). Also, for elementary tgds, in case op_2 is simply a multi-tuple operator, ϕ_2 is absent, hence z is the only argument of K_2.

[8] As in the rest of the paper, we refer to tgds with one atom in the rhs.

Here we assume that operators do not act on dimensions as it would not add much to the general discussion, however in Section 5.5 we will show that we cover this case as well in the system.

We define the *composition* of two consecutive tgds (in the general form (1) shown above), with $\text{op}_1 = \text{op}_1^M \circ \text{op}_1^T$ and $\text{op}_2 = \text{op}_2^M \circ \text{op}_2^T$, as follows.

$$\begin{aligned}\sigma_2 \circ \sigma_1 : \phi_2(\cdot, z_1, \ldots, z_m) \wedge \phi_1(\cdot, y_1, \ldots, y_n) \rightarrow \\ K_2(\cdot, (\text{op}_2^M \circ \text{op}_1^M \circ \text{op}_2^T)(z_1, \ldots, z_m, \text{op}_1^T(y_1, \ldots, y_n))\end{aligned} \quad (2)$$

Two comments are useful. First, as it will turn out that this definition leads to effective composition in some cases and has an undesirable behavior in others—we will distinguish between the two situations by means of the notion of correctness. Second, the definition refers to the general case where each operator has both a tuple-level and a multi-tuple constituent. To illustrate it, we first refer to the base cases with only one constituent per operator and then to the general case.

Let us consider the base case where each operator $\text{op} = \text{op}^M \circ \text{op}^T$ is either simply tuple-level (and reduces to op^T) or simply multi-tuple (it reduces to op^M).

op_2 / op_1	tuple-level	multi-tuple
tuple-level	$K_2(\cdot, \text{op}_2(z_1, \ldots, z_m, \text{op}_1(y_1, \ldots, y_n))$	$K_2(\cdot, \text{op}_2(\text{op}_1(y_1, \ldots, y_n))$
multi-tuple	$K_2(\cdot, \text{op}_1(\text{op}_2(z_1, \ldots, z_m, y_1)$	$K_2(\cdot, \text{op}_2(\text{op}_1(y_1))$

Fig. 4: Simplifications of the rhs of the composition for the base case.

Figure 4 shows how the rhs of (2) is simplified in the various cases. If op_1 is tuple-level, then op_1^M does not appear (technically, it is the identity operator) and so op_2 is the outermost operator, independently of its tuple-level or multi-tuple nature. We first calculate op_1 on its parameters $y_1, \ldots, y_n$ and then use the result value as a parameter (z in σ_2) for op_2.

If op_1 is multi-tuple, then y_1 is its only parameter. Thus, if op_2 is multi-tuple as well, then op_1 will be the innermost and its result will be the only parameter of op_2.[9] Otherwise, if op_2 is tuple-level, op_1 will be the outermost.

Observe that with respect to the orderly application of dependencies σ_1 and σ_2, in the multi-tuple/tuple-level case the operators are nested in the reverse order. In particular, op_2^T is applied between op_1^T and op_1^M, breaking the original order. This inversion is essential and unavoidable in the definition for type reasons, as it guarantees that the multi-tuple constituents of the operators

[9] Indeed, the case where the two operators are multi-tuple and have different grouping dimensions requires a slight extension of the syntax, where the grouping dimensions would be specified as an argument of the operator itself, so, for example $R(x, y, z) \rightarrow Q(x, \max(\text{avg}(z, \text{group by } x, y)))$, calculates the maximum, grouped by x, of the averages of z, grouped by x and y.

are always the outermost. In this way, we are also guaranteed that in the rhs there are no variables whose values are not generated by a grouping function, whenever a grouping occurs. Otherwise, if we allowed variables external to the grouping functions, their values would be non-deterministically chosen among those of the bag of values of each group and so the tgd would be ill-defined. As we will see, this "inversion" is a crucial point and makes the correctness of tgd composition a non-trivial issue.

Let us now come to the general case, in which op_1 and op_2 have both tuple-level and multi-tuple constituents, for example as a result of previous compositions. Our definition of composition is *closed* with respect to our language for tgds: a composition always produces tgds to which composition can be applied. The closure property can be easily verified by showing that (2) has the same form as the tgds used for σ_1 or σ_2 in (1). The lhs of (2) contains a conjunction of atoms, coherently with (1). The rhs of (2) can be rewritten into an equivalent way as follows:

$$K_2(\cdot, (\mathrm{op}_2^M \circ \mathrm{op}_1^M \circ \mathrm{op}_2^T \circ \mathrm{op}_1^{T\prime})(z_1, \ldots, z_m, y_1, \ldots, y_n)) \tag{3}$$

In (3), operator $\mathrm{op}_1^{T\prime}$ is the m+n-ary extension of op_1^T that takes as input $z_1, \ldots, z_m, y_1, \ldots, y_n$ and produces the same results as op_1^T for $y_1, \ldots, y_n$ and simply ignores $z_1, \ldots, z_m$. Then, for the associativity of composition, we have that $\mathrm{op}_2^M \circ \mathrm{op}_1^M$ and $\mathrm{op}_2^T \circ \mathrm{op}_1^{T\prime}$ respectively represent the multi-tuple and the tuple-level constituents of a new generic operator. Thus, like in (1), the rhs of (2) is the application of a generic operator. Therefore the closure property is respected.

We also argue that our composition is *associative*, that is, given three pairwise consecutive tgds σ_1, σ_2 and σ_3, we have that the composition $(\sigma_1 \circ \sigma_2) \circ \sigma_3$ produces the same tgd as $\sigma_1 \circ (\sigma_2 \circ \sigma_3)$. This can be proven in a straightforward way on the basis of the associativity of operator composition.

Let us now see our definition of composition in action with some examples:

(a) $S(x, s), T(x, t) \rightarrow V(x, s + t)$,
(b) $R(x, r), V(x, v) \rightarrow Q(v, r + v)$

The composition of the two above dependencies is
$R(x, r), S(x, s), T(x, t) \rightarrow Q(x, r + (s + t))$. This is a very initial case, since the tgds include only tuple-level operators. In practice, we replaced $V(x, v)$ in (b) with the conjunction $S(x, s), T(x, t)$ from (a). In (a) and (b) the variables denoting the dimensions have been unified (into x) and a renaming has been applied to avoid ambiguities among the variables denoting the measures.

A more sophisticated case, which involves a multi-tuple operator in the first tgd, follows:

(m) $S(x, y, z) \rightarrow V(x, \mathrm{sum}(z))$
(n) $R(x, k, w), V(x, q) \rightarrow Q(x, w \times q)$

Applying the definition of composition, we have $n \circ m : R(x, k, w), S(x, y, z) \rightarrow Q(x, \mathrm{sum}(w \times q))$. Observe that if we orderly apply (m) and (n), we first have the summation and then the multiplication. Instead, in $n \circ m$, the operations are applied in the reverse order. Interestingly, as the results in the next Section will motivate, we obtain the same outcome. Consider for example an input instance with the facts $S : \{(1, 0, 2), (1, 2, 3), (2, 3, 1)\}$, $R : \{(1, 0, 2), (2, 3, 4)\}$ and notice that in both the ordered application and the composition, the outcome is $Q : \{(1, 10), (2, 4)\}$. It is also important to outline that in composition $n \circ m$, the operator $\mathrm{sum}(w \times z)$ has both a multi-tuple and a tuple-level constituent (sum and $w \times q$, respectively). Let us consider the following tgd:

$(p)\ Q(x, y), Z(x, z) \rightarrow P(x, y \times z)$

Now, since $n \circ m$ and p are consecutive, we can calculate $p \circ n \circ m$, which is $R(x, k, w), S(x, y, z), Z(x, z) \rightarrow P(x, \mathrm{sum}(w \times q \times z))$. For example, for $Z : \{(1, 5), (2, 2)\}$, we would have $P : \{(1, 50), (2, 8)\}$. Also observe that we would obtain the same result tgd, if we first composed $p \circ n$ and then $p \circ n \circ m$.

5.5 Correctness of composition

In order to be exploited for optimization, the application of a composed tgd to a source instance should give the same results as the orderly application of the two tgds. In this Section, we study the general problem of composing two consecutive tgds in the presence of operators of any kind. We define the notion of *correctness* for the composition and characterize the *composability* problem by presenting a necessary and sufficient condition for it. Finally, we show that in our context, the composability of the tgds can be verified in a simple way, just with reference to the algebraic properties of the involved operators.

What makes the problem of composing tgds in the presence of operators interesting and non-trivial is that two tgds are not always composable as it may be the case that the results given by the composed tgd differ from the ones obtained with the orderly application of the two. Consider, for example, the following setting:

$(h)\ S(x, y, z) \rightarrow V(x, \mathrm{sum}(z))$
$(k)\ R(x, k, w), V(x, q) \rightarrow Q(x, w + q)$.

If we apply the definition we have given in Section 5.4, we obtain the following composed tgd $R(x, k, w), S(x, y, z), \rightarrow Q(x, \mathrm{sum}(w + z))$. Let us now consider a source instance with the following facts $S : \{(1, 0, 2), (1, 2, 3), (2, 3, 1)\}$, $R : \{(1, 0, 2), (2, 3, 4)\}$. If we orderly apply (h) and (k), we obtain $Q : \{(1, 7), (2, 5)\}$. Conversely the composition returns a different result instance $Q : \{(1, 9), (2, 5)\}$.

Given two consecutive tgds σ_1 and σ_2, we say that their composition $\sigma_1 \circ \sigma_2$ is *correct* if for every pair $\langle I, J\rangle$ that satisfies $\sigma_1 \circ \sigma_2$, there exists an instance K, such that $\langle I, K\rangle$ satisfies σ_1 and $\langle K, J\rangle$ satisfies σ_2 and vice versa.

The following result characterizes the problem of *composability*, that is, it tells whether two consecutive tgds can be composed.

Let us first introduce a preliminary notion. We say that there exists a *single cardinality constraint* between a nonempty subset $x_1, \ldots, x_k$ of the common attributes of two relations R_1 and R_2 if for every tuple $(x_1, \ldots, x_k, \ldots, x_l)$ of R_1 there is at most one tuple $(x_1, \ldots, x_k, \ldots, x_s)$ in R_2 with the same values for $x_1, \ldots, x_k$.

Theorem 3 *Given two consecutive tgds:*

– $\sigma_1 : \phi_1(\cdot, y_1, \ldots, y_n) \rightarrow K_1(\cdot, (op_1^M \circ op_1^T)(y_1, \ldots, y_n))$
– $\sigma_2 : \phi_2(\cdot, z_1, \ldots, z_m) \wedge K_1(\cdot, z) \rightarrow K_2(\cdot, (op_2^M \circ op_2^T)(z_1, \ldots, z_m, z))$

their composition $\sigma_1 \circ \sigma_2$ is correct (and we say that σ_1 and σ_2 are composable) if and only if the following two conditions hold:

(a) the grouping operation in op_1^M involves a nonempty subset of the dimensions of ϕ_1 such that there exists a single cardinality constraint from those dimensions to a subset of the dimensions of ϕ_2;
(b) op_2^T distributes over op_1^M.

Proof Applying our definition of composition given in Section 5.4, definition (2), and rewriting the rhs as explained in (3) in the same Section, we have:

$$\sigma_2 \circ \sigma_1 : \phi_2(\cdot, z_1, \ldots, z_m) \wedge \phi_1(\cdot, y_1, \ldots, y_n) \rightarrow \\ K_2(\cdot, (\mathrm{op}_2^M \circ \mathrm{op}_1^M \circ \mathrm{op}_2^T \circ \mathrm{op}_1^{T\prime})(z_1, \ldots, z_m, y_1, \ldots, y_n)) \quad (4)$$

Let K be the result of applying σ_1 to a generic instance I and let J be the result of applying σ_2 to K. Because σ_1 and σ_2 are full, we have that K and J are unique. Then, let J' be the result of the application of $\sigma_2 \circ \sigma_1$ to I, unique as well.

We want to show that if conditions (a) and (b) hold, then the composition is correct, that is $J = J'$ (sufficient condition) and that there are no other cases in which the composition is correct (necessary condition), that is, in every other case $J \neq J'$.

Let us start with the sufficient condition. Since there is a cardinality constraint from the dimensions of ϕ_1 to those of ϕ_2 and the aggregation in op_1^M preserves all the dimensions of ϕ_1 that participate in such cardinality constraint (by condition (a)) and op_1^T does not alter the values of such dimensions, then the dimension tuples in K that derive from coalescing into the same group those of I in σ_1 unify at most once in the join $\phi_2 \wedge K_1$ in σ_2 and the corresponding dimension tuples of I unify at most once in $\phi_2 \wedge \phi_1$ in (4). Moreover, the dimension tuples of K that are discarded in $\phi_2 \wedge K_1$ in σ_2, correspond to dimension tuples of I that are discarded in $\phi_1 \wedge \phi_2$ in (4). Intuitively, when (a) holds, the grouping in op_1^M can be applied indifferently before or after the join, resulting in the same number of tuples, with the same values for the dimensions. As for the measures, in (4), we apply the following composition to I in order to obtain J': $\mathrm{op}_2^M \circ \mathrm{op}_1^M \circ \mathrm{op}_2^T \circ \mathrm{op}_1^T$, while in the orderly application of σ_1 and σ_2, we apply $\mathrm{op}_1^M \circ \mathrm{op}_1^{T\prime}$ to I to obtain K and then $\mathrm{op}_2^M \circ \mathrm{op}_2^T$ to K to obtain J. As we have said in Section 5.4, $\mathrm{op}_1^{T\prime}$ produces the same measures as op_1^T; hence, let L be the result of applying either of them to I. Then, we want to prove that for L, applying $\mathrm{op}_1^M \circ \mathrm{op}_2^T$ equals to applying $\mathrm{op}_2^T \circ \mathrm{op}_1^M$. Recalling the semantics of multi-tuple operators, we have that op_1^M applies to a multiplicity of tuples in L and combines them, whereas op_2^T performs a tuple-level combination of the measures of such tuples. Since by condition (b), op_2^T distributes over op_1^M, both the compositions produce the same output instance N. Therefore, because both in (4) and in σ_1 and σ_2 we apply op_2^M to N, it follows $J = J'$.

For the necessary condition, we show that whatever violation to either condition (a) or (b) leads to an incorrect composition ($J \neq J'$). For condition (a), let us suppose that there is no single cardinality constraint from any subset of the dimensions of ϕ_1 to the dimensions of ϕ_2 that is preserved by the grouping operation in op_1^M. In this case, given an instance I, if we apply σ_1, there is at least one dimension tuple in K, deriving from coalescing into the same group those of I, for which the corresponding tuples in I match with more than one tuples in the join $\phi_2 \wedge \phi_1$. Therefore, in (4), operator op_1^M considers twice at least a dimension tuple that in σ_2 is considered once, resulting in $J \neq J'$.[10] To violate condition (b), it is sufficient to consider any two mathematical operations such that the distributivity does not hold to show that the values of the calculated measures are not correct. In effects, in these cases, distributing the application of op_2^T in op_1^M is not correct by definition. A typical example involves summation and sum: $\sum_i x_i + c \neq \sum_i (x_i + c)$. $\square$

The above result characterizes the problem of telling whether two tgds are composable in the presence of generic operators. However, while condition (b) can be statically verified in constant time since it only depends on the algebraic properties of the involved operators, condition (a) is data dependent and exponential time in the number of the involved dimensions. In EXLEngine, we consider a more specific context, very common in practice, where tgd composability can be verified only on the algebraic properties of the operators.

So far in this Section, we have considered the most general form of n-ary tuple-level operators in tgds, with no assumptions on their dimensions. Let us now assume that n-ary tuple-level operators always involve operands such that the dimensions of (at least) one are a superset of the dimensions of each of the others (and all those dimensions appear in the result).

With this restriction, we can derive the following theorem.

[10] Notice that there is the residual, and indeed remote, possibility in which the repeated dimension tuple has the identity element as its measure for the aggregation under consideration or that, in general, the repeated tuples compensate the error. However this condition is value and aggregation dependent and should be considered as a case of "correctness by chance".

Theorem 4 *Two consecutive tgds σ_1 and σ_2 such that the dimensions of K_1 are a superset of those of ϕ_2 are composable if and only if op_2^T distributes over op_1^M.*

Proof Let us consider the two tgds. Since by hypothesis the dimensions of K_1 are a superset of those of ϕ_2 (or ϕ_2 is not present, in which case the condition is trivially satisfied), then in σ_2, each tuple in K_1 unifies with at most one tuple in ϕ_2. Since K_1 derives from ϕ_1 in σ_2, then ϕ_1 has a superset of the dimensions of K_1 and thus a superset of the dimensions of ϕ_2 as well. Therefore, each tuple of ϕ_1 unifies at most with one tuple of ϕ_2, satisfying (a). Thus, σ_1 and σ_2 are composable if and only if (b) holds. $\square$

This result is particularly useful in practice, and adopted in EXLEngine, because the superset-subset verification is polynomial in the number of the dimensions and the distributivity check is constant time. Indeed, we are aware that by applying this condition in the system, we choose not to compose the consecutive tgds such that condition (a) holds although K_1 has a subset of the dimensions of ϕ_2. However, these cases are rather rare since, in most of the statistical programs, the operand relations in a tuple-level operator have exactly the same dimensions and aggregations do not entail projecting dimensions out, but just changing their grain, for example as we have seen for operators on dates.

A final remark concerns the possible modifications to dimensions. Notice that we have assumed that op_1 does not alter the values of the dimensions. Indeed, in Section 4 we have introduced such possibility, for example with operators acting on dates, such as *shift* and *quarter*. We handle these cases with an intuitive extension of the definition of composition we have given in Section 5.4, which can be explained by the following example:

(l) $A(t,x) \rightarrow B(t-1,x)$
(m) $B(t,y), C(t,z) \rightarrow E(t,y+z)$.

The two above tgds are consecutive, but (l) has an operator that modifies the time dimension and no operators that modify the measures. We assume (l) has the identity operator (which does not alter the measure) and account for the shift in the join condition so that we can calculate $m \circ l$ with the usual definition: $A(t-1,x), C(t,z) \rightarrow E(t,x+z)$.

5.6 Optimization by composition

Once it has generated one tgd for each elementary statement, EXLEngine builds up a tgd graph $\mathcal{G}$ for it. A *tgd graph* $\mathcal{G}(V,E,\lambda)$ for a schema mapping $M(\mathbf{S},\mathbf{T},\Sigma)$ is a graph, having a vertex M_i in V for each tgd σ_i of Σ and an edge (M_i,M_j) for each pair of vertices that correspond to consecutive tgds (σ_i,σ_j) of Σ. Each vertex M_i is labeled with a string representing its respective tgd σ_i.

In the graph, every internal vertex represents an intermediate result of the calculation, which is then used by the subsequent vertices as an input. Final results are represented by leaf vertices (outdegree zero), while roots (indegree 0) stand for the elementary relations.

We want to modify and compact the graph so as to delegate the optimization responsibility to the target system. This is based on the assumption that sharing intermediate temporary results is more efficient than having many separate, locally optimized, tgds. In general this is questionable, but appears reasonable in a heterogeneous context, where we deal with a plenty of different target systems. We are aware that a runtime optimizer adopting cost-based techniques to detect common subqueries could achieve good performances in some cases, for example in the presence of small volumes of data, yet this subquery detection functionality is unavailable in the prominent statistical tools, and only to a very limited extent even in relational systems.

Let us see how we proceed in detail. We search for all the composable pairs of tgds such that the result of the first is used only by the second and coalesce them into a single tgd. In case the result of a tgd is used by more than one tgds, we choose to ignore them and do not compose multiple times.[11]

Let us now come to the details of our optimization algorithm. By construction, $\mathcal{G}$ is acyclic and, for each vertex, three kinds of subgraph structures are possible, as shown in Figure 5:

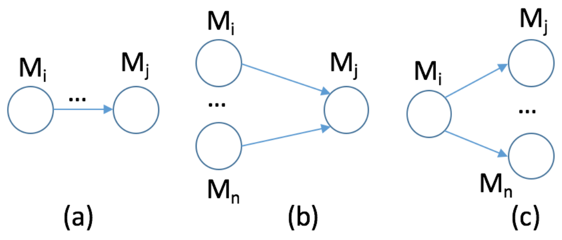


Fig. 5: (a) a sequence; (b) a convergence; (c) a divergence.

(a) *sequences*, when an edge links two vertices M_i, with one outgoing edge (outdegree is one), and M_j, with one incoming edge (indegree is one); the tuples produced by M_i are used only by M_j, therefore we choose to compose the two dependencies into one; (b) *convergences*, when the indegree of M_j is greater than one; also in this case, the tuples produced by each incoming M_i are used only by M_j, therefore we choose to compose $M_i, \ldots, M_n$ and M_j into one tgd; (c) *divergences*, when the outdegree of M_i is greater than one. In this case the tuples produced by M_i are used my multiple dependencies. As explained,

[11] There is the residual possibility that two EXL statements share a subsexpression, resulting in two tgds sharing one ore more atoms of the lhs. Since we break down all the statements into elementary statements, we could end up having tgds with coinciding premises, which indeed we detect and simplify in the system.

our choice here is not composing M_i and $M_j, \ldots, M_n$ and considering M_i as a *materialization point*, so as to maximize the reuse of common results.

Starting from a vertex M_i, EXLEngine performs a depth-first traversal, analyzing the successors. As long as we are in the presence of either a sequence or a convergence centered in M_i, the algorithm explores all the successors and keeps on compacting them. In case of divergence, a fresh depth-first traversal recursively starts from every "undiscovered" successor.

As any depth-first traversal, the algorithm always terminates in at most $|V|+|E|$ steps. The correctness of the algorithm is guaranteed by the composability of the tgds that are compacted and by the associativity property of out composition, which ensures that whatever composition succession is equivalent (and correct).

Figure 6 shows the tgd graph, built for the mapping in our running example and Figure 7 is its optimized version. Dependencies (1a), (1b), (2) and (3) form a sequence, which has been compacted into (1.2.3). Tgd (4) and (1.2.3) have not been compacted since not composable; similarly for (4) and (5.6.7) ((5) has been obtained by rebuilding (5a-d)). Tgd (7), obtained by rebuilding (7a) and (7b) and has been composed with (6) and (7). Tgd (1.2.3) produces a divergence and is not composed anymore.

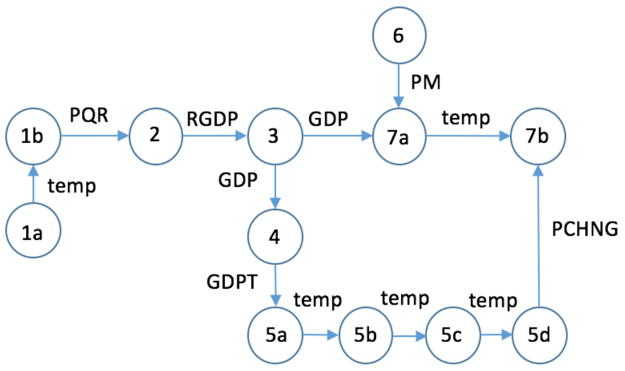


Fig. 6: The tgd graph of the mapping of our running example. Nodes are labelled with the progressive number of the dependency (see Section 2), and edges with the atoms in common between consecutive dependencies. Labels "temp" denote a temporary anonymous result, deriving from the decomposition of multi-operator statements.

Let us now come again to our running example and see how the mapping graph in Figure 6 can be reduced to fewer composed mappings (depicted in Figure 7) with our algorithm.

Starting from node (1a), as its outdegree is 1, we check if it is can be composed with (1b) and rebuild the original tgd (1) (and we do the same for all the decomposed tgds). Now, as reported in Section 2, tgd (1) aggregates by averaging over p and groups by quarter(d) and r; tgd (2) uses the result PQR. We join along q and r and multiply the measures. As the multiplication distributes

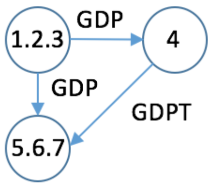


Fig. 7: The reduced tgd graph.

over the average ($\text{avg}(n) \times c = \text{avg}(n \times c)$) and PQR and RGDPPC have the same dimensions, the two mappings are composed into (1.2) as follows:

$$\text{PDR}(q, r, p) \wedge \text{RGDPPC}(q, r, g) \rightarrow$$
$$\text{RGDP}(\text{quarter}(q), r, \text{avg}(p * g))$$

Then, we need to check if (1.2) can be composed with (3). They share RGDP and tgd (3) aggregates by sum grouping by q. As the second operator does not have tuple-level constituent, we can simply compose by nesting them. We obtain tgd (1.2.3):

$$\text{PDR}(q, r, p) \wedge \text{RGDPPC}(q, r, g) \rightarrow$$
$$\text{GDP}(q, \text{sum}(\text{avg}(p * g, \text{group by quarter}(q), r)))$$

Now, tgd (1.2.3) shows a divergence, as both tgd (4) and (7) use GDP. Therefore, we move to (4). Tgds (4) and (5) share GDPT, but (4) has an aggregation stl_T. Since arithmetics in (5) are not distributable with respect to seasonal decomposition, (4) and (5) are not composed. In particular, (4) is unchanged:

$$\text{GDP(q, r)} \rightarrow \text{GDPT(q, stl\_T(r))))}$$

We move to (5), and check if it is composable with (7). We just have tuple-level operators in both the mappings, therefore no particular conditions need to be checked. The same applies to (6) and we have the composed mapping (5.6.7):

$$\text{GDP}(q, r_1) \wedge \text{PM}(q, r_2) \wedge \text{GDPT}(q, r_3) \wedge \text{GDPT}(q - 1, r_4)$$
$$\rightarrow \text{FGDP}(q, r_1 \times r_2 \times ((r_3 - r_4) \times 100/r_3)).$$

6 Generating Executable Code

The final step in our process is the translation of the schema mapping (generated and optimized as shown in Section 5) into a form that can be executed on target systems. This is the goal of the present section. As we said in the introduction, we are interested in various implementations, which include relational DBMSs as well as domain-specific tools, such as R and Matlab, two widespread domain-specific tools, and also ETL flows as used in business intelligence processes. In Section 6.1 we describe the general method, while in Section 6.2 we see translations towards specific target systems.

6.1 The method

The translation is based on the observation we made in Section 5.2 that the tgds in our schema mappings can be applied orderly one at a time. The result of each mapping is then the input of the following ones. So, without loss of generality, we can concentrate on the translation of each tgd independently of the others.

As explained in Section 5.1, our tgds have, in the rhs, a single atom and, in the lhs, the conjunction of two or more atoms, or a single atom. From a procedural point of view, each tgd can be satisfied with application of a fixed series of *actions*. Relations in the lhs are always assumed to be materialized (and non-composable mappings materialize their rhs). Hence, input relations are retrieved from the storage system, by means of the atoms in the lhs (*get* actions); if there is a conjunction $\wedge$ in the lhs, the relations are joined (*merge* action). Computations are then specified in the rhs (*calculation* action) and finally the result is "saved" into the result relation (*assignment* action). Each action generates an intermediate relation, which is then used as the input for the subsequent actions. For example, in case of three relations in the lhs, the merge action joins them. Calculations are then performed and the result is finally stored into the system. In many languages this behavior can be obtained by the use of temporary variables (as it happens with R and Matlab) storing intermediate relations; in SQL we adopt nested queries, although we could have also used views or temporary tables as well; in ETL tools, the generated steps are simply connected in the appropriate order.

For example, we now consider four of the tgds we have seen in Section 5.1, after the optimization phase.

(1.2.3) $\text{PDR}(q, r, p) \wedge \text{RGDPPC}(q, r, g) \rightarrow$
$\qquad\qquad \text{GDP}(\text{quarter}(q), r, \text{sum}(\text{avg}(p * g)))$
(4) $\text{GDP} \rightarrow \ \text{GDPT}(\text{stl\_T}(\text{GDP}))$

In terms of actions, with a syntax which is mainly self-explanatory, they would be expressed as follows:

(1.2.3)

```
PDR = get(PDR, [q, r], p);
RGDPPC = get(RGDPPC, [q, r], g);
MERGED = merge(PDR, RGDPPC);
CALC = calculate(MERGED, [multiply, avg, sum], groupBy = [q];
save(CALC, GDP);
```

(4)

```
GDP = get(GDP, [q], g);
CALC = calculate(PQR, stl_t)
save(CALC, GDPT);
```

Now, apart from specific calculations, all target systems have features that implement actions of these types (which we do not describe here for the sake of space). As a consequence, our tool can generate various executable versions, one for each target system, with the only exceptions related to missing computation operators. The algorithm is *system-independent* but *system-aware*: it is system-independent, since its logic does not change, whatever the target system; it is system-aware, in the sense that the algorithm knows the syntax of the target systems and generates the executable statements accordingly. However, the only system-dependent part is the generation of executable code out of the actions.

6.2 Translation examples

We now present the complete translations generated by our method and tool for the composed tgds shown in Section 6. We consider the various target systems one at a time.

6.2.1 SQL mappings

Let us consider tgd (1.2.3), which is quite general since it includes both tuple-level and multi-tuple operators. Our tool translates it into an SQL insertion statement as follows.

```
INSERT INTO GDP(Q,R,P)
 SELECT to_quarter(MERGED.Q) AS Q, MERGED.R AS R,
        AVG(SUM(MERGED.P * MERGED.G)) AS P
 FROM
 SELECT * FROM (
(SELECT PDR.Q AS Q, PDR.R, PDR.P
 FROM
   (SELECT Q,R,P FROM PQR_TABLE) PDR,
   (SELECT Q,R,G FROM RGDPPC_TABLE) RGDPPC
 WHERE PDR.Q = RGDPPC.Q
   AND PQR.R =  RGDPPC.R
 ) MERGED
 GROUP BY Q, R
```

The atoms in the lhs lead to two *get* subqueries, which are generated first. They are then combined with the join on the corresponding aliases PDR and RGDPPC (in the *merge* action), with the equality condition for the repeated variables in the lhs. Measures are combined in a *calculation* action by means of a tuple-level and multi-tuple operators and the result is inserted, with an *assignment*, into the target table.

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. All the calculations appear, properly nested, in the target list. In this query, the aggregation is absolutely similar to those common in SQL and the generation of the executable statement is straightforward and it includes a GROUP BY clause as well as SUM and AVG.

Tgds with complex multi-tuple operators require more care. Let us consider tgd (4). It does not reduce the cardinality by partitioning and combining tuples, but essentially computes a new table where each tuple depends on several (possibly all) of the tuples in the argument. Thus for tgd (4), the corresponding *calculation* action uses a tabular function, and we have:

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

In the script, the tabular function `STL_T` returns the trend component for a given time series stored in a table with suitable 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.

6.2.2 Mappings in specialized languages

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

We 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 (1.2.3) requires various statements.

```
merged <- merge(PDR,RGDPPC,by=c("q","r"))
calculated$i <- merged["p"] * merged["g"]
calculated <- calculated[-c("p","g")]
calculated$q <- to_quarter(calculated$q)
calculated <- aggregate(
   aggregate(calculated, by=list(c(q,r)),FUN="sum"),
      by=list(c(q,r)), FUN="avg")
```

Get actions are omitted, since they directly correspond to the names of the relations.

The first statement derives from the *merge* action and uses the R operator called *merge* to build a temporary matrix by joining PDR and RGDPPC on q and r. The second statement adds a new column i to `merged`, computed as the element-wise product of p and g (*calculation*). Then original columns p and g are removed (*assignment*), and finally sum and average are calculated after of the conversion of q to quarters.

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.

```
joined=join(PDR, 1:2, RGDPPC, 1:2)
calculated[;5]=joined[ ; 3] .* joined[ ; 4]
calculated=[calculated[ ; 1] calculated [ ; 2] calculated [ ; 5]]
calculated[;1]=to_quarter(calculated[;1])
GDP = grpstats(grpstats(calculated, {1,2},
      {'sum'}),{1,2},{'mean'})
```

Get actions are trivial again. Here the Matlab *join* operator (*merge* action) is used to build the temporary matrix `joined`, as the composition of PDR and RGDPPC; its four columns correspond, in order, to variables q, r, p and g in the tgd. Then, the second statement (corresponding to a *calculation* action) adds a new column, i (in position 5), as the element-wise product of p and g (here denoted by positions 3 and 4, respectively). The subsequent assignment keeps only the needed columns. Then time is converted to quarters and, finally, the last statement (*assignment*) builds GDP by calculating the appropriate aggregations.

6.2.3 ETL translation

ETL and Business Intelligence tools are another common family of target systems. The translation engine can indeed turn schema mappings into executable ETL jobs by feeding the metadata catalog of the specific tool, where the jobs are described. Many tools provide APIs to interact with the catalog and build the ETL jobs programmatically. Other tools require a more complex interaction, e.g., through XML files or command-line interfaces. EXLEngine actually supports Pentaho Data Integration,[12] 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 the total order of tgds. For each atom in the lhs, a *data source step* appears in the flow as the translation of *get* actions. The input streams coming from the data source steps are joined (*merge*) on the dimensions by means of 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* (for the corresponding action), which actually implements the rhs of the tgd. Finally, the stream is written back into the system with an *output step* (*assignment*).

With reference to tgd (1.2,3), the ETL flow in Figure 8 is generated.

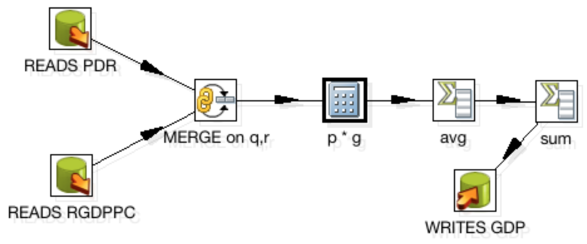


Fig. 8: Example of schema mapping deployment as ETL flow.

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

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.

7 Implementation: EXLEngine

The approach to statistical data processing described in this paper has been concretely embraced in developing EXLEngine, the calculation engine adopted by the Bank of Italy. In this system, programs are written in EXL, the language we illustrated in Section 4 and translations towards most popular systems for statistics are implemented as shown in Section 6.1.

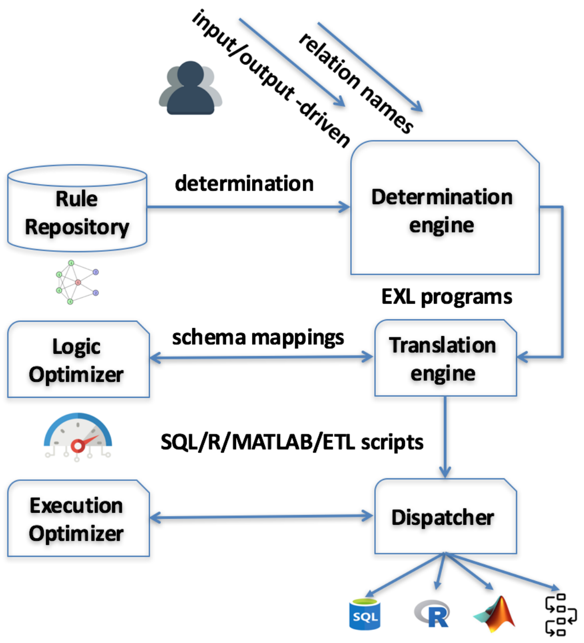


Fig. 9: The architecture of EXLEngine.

The typical interaction with the engine are shown in Figure 9. EXLEngine stores all the statements of all the available EXL programs in a *rule repository*. Each statement defines a set of dependencies between the relations in the lhs and the one in the rhs. The repository can be seen as a network of dependencies, or a *knowledge graph*, where the nodes are the relations and an edge represents the fact that the successor is calculated from the predecessor. The tuples of each node are calculated by combining all the adjacent predecessors in the graph, according to the specific EXL statement that encodes how to combine them. Given such a graph, two tasks can be performed: 1.

input-driven calculation: given some input relations, we want to update all the descendant relations in the graph; 2. *output-driven calculation*, i.e., given some output relations, we want to update all their tuples from the values of the predecessors. These two tasks can be also seen as forward- vs backward-chaining approaches typical in logic programming or, in another perspective, as materialization of the data exchange results vs query answering.

So, given a set of names for relations and the input-driven/output-driven mode, the *determination engine* builds at runtime an EXL program to be run, by extracting the necessary dependencies from the repository.

The *translation engine* performs the schema mapping generation algorithm presented in Section 5 and produces a translation for each tgd. The translation engine interacts with a *logic optimizer*, which executes the optimization algorithm and packs the generated tgds into fewer and more compact ones. The translation engine finally produces the executable scripts, suitable for the specific target systems.

At runtime, the *dispatcher* component delivers the scripts to the target systems. To this end, the obtained tgds are grouped by an *execution optimizer* into clusters whose relations can be independently processed by different *target engines*. Each target engine then executes only its native code and produces the results. For example a cluster may be calculated by a RDBMS that receives a SQL translation, while another one may be best suited for an R node. The interaction with the actual back-end tools takes place via dedicated adapters to the tools' APIs (e.g., JDBC, R Java interface, etc.).

7.1 System status and performance

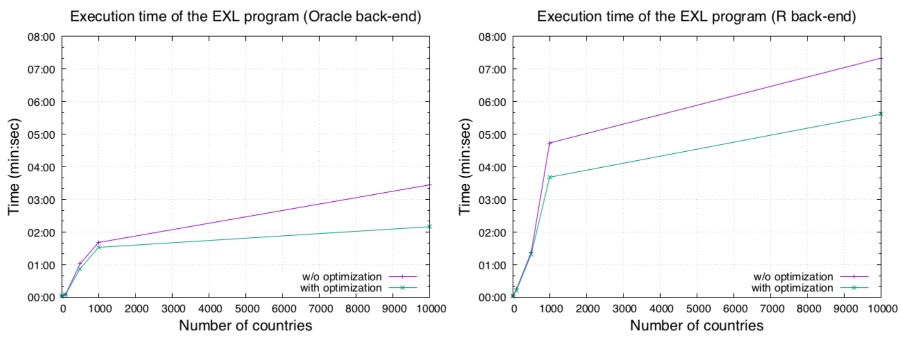


Fig. 10: Execution times for EXLEngine.

The system at Bank of Italy fully implements the described tgd language with a number of statistical operators and is used in many real applications. Many extensions, especially those regarding the interaction of different back-ends with one another, are under development and will be integrated in the future. The application of our approach in many business scenarios shows good performance and we plan to conduct a full-scale evaluation soon. Results in Figure 10 give a glimpse on the current performance in a synthetic case that we built to test the system at scale. We calculated the GDP for a number of fictional countries, executing the calculation with Oracle and R back-ends. Results are promising and show the effectiveness of our optimization techniques based on composition, especially in the presence of advanced optimizers.

8 Related Work

This section describes prior work on the field and alternative approaches. First we recall some foundational approaches motivating and inspiring EXLEngine; then we discuss alternatives for scaling in statistical data processing.

Bridging the gap between specification and execution

EXLEngine strongly relies on the presence of an intermediate level, that of schema mappings, decoupling the high-level specification from the executable code and supporting optimization. Schema mappings and their properties have been analyzed in detail in a number of works [8]. Here we mostly use Fagin et al.'s [26] definitions and adopt their language for tgds with some extensions. Although we propose some modifications to support our needs (introduction of mathematical and statistical operators), we do not alter the declarative essence of the tgd language, and the resolution methods (e.g., the chase) remain applicable. For data exchange, we refer to classical formulations [1].

Composition of generic schema mappings has been originally characterized by Fagin et al. [28] and more recently extended in the presence of target constraints [2]. Various application settings for composition have been proposed (e.g., schema evolution [29]), including mapping optimization [30]. With this paper, we contribute with a further application of mapping compositions, analyzing the internal characteristics of operators, discussing their composability, and actively using them in the optimization.

The translation-based strategy of EXLEngine leverages the experience of Atzeni et al. [6,7,5] 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 do not foster a pivot metamodel but directly use mappings as a hub between an abstract transformation specification and its implementation.

We make specific reference to the Matrix data model and to the EXL language [21], both developed by the Bank of Italy, but any other language corresponding to the same logic fragment could have been considered.

Approaches and algorithms for the execution of schema mappings in SQL systems and DBMSs have been proposed in several works and in EXLEngine we exploit these experiences to build a system-independent and multi-target translation engine, able to address the most adopted tools in statistics.

A pioneer system to execute schema mappings is *Clio* [25,31], which provides semi-automatic schema mapping generation features as well as mechanisms to translate graphical mappings into executable queries. Also *+Spicy* [35] delivers a system translating schema mappings into executable queries, not only encoded in SQL but also in XQuery. In this sense, it is one of the first systems to decouple a high-level specification from the executable statements. Unlike EXLEngine, Clio and +Spicy directly rely on the mappings, described within a graphical CASE tool, for the specification; conversely, we choose to introduce a user-oriented specification language to make the system more suitable for a real production context. In facts, according to our experience, graphical environments tend to become unproductive when dealing with tens or hundreds of objects (hence even thousands of dependencies), as this is the case for statistical data processing. On the other hand, other systems, such as MIDST [5] and *HePToX* [13], directly use a logic-based language to describe mappings. In this sense, we could directly feed schema mappings into EXLEngine, as the first correspondence with the initial EXL statements is rather intuitive. However, we propose a programming-like specification language, as it is more user-friendly and easier to grasp.

Scaling in statistical data processing

Many existing schema mapping systems (Clio, +Spicy, HepTox, MIDST) are based on a one-to-one correspondence between mappings and queries. Indeed, the user specifies the mapping with some formalism, and the tool produces an executable translation for some target system. This practice impedes an effective use of the optimizers of such systems and heavily relies on a good mapping design. EXLEngine actively exploits and manipulates schema mappings to support multi-query optimization in a system-independent way.

With respect to mapping execution, an important category of related systems are the chase-based tools [10], designed to enforce a set of constraints (e.g., in the form of tgds) over a database instance for several goals, including data exchange and data cleaning. Native implementations of most advanced chase techniques often lead to very good performance, although their general purpose target and absence of domain-specific features (for example many even lack algebraic operations) make an evaluation over the statistical setting not promising. On the other hand, reasoners and knowledge graph management systems [9] promise to be important players in scalable data processing, and the interaction with external systems (for example specialized statistical and machine learning tools) is considered among the core desiderata; thus, a convergence of the efforts would lead to interesting developments.

The notion of composition we introduce for optimization has some commonalities with the classical techniques adopted in query-optimization problems [17,18] and it is our opinion that our results also generalize the *view unfolding* problem, giving a necessary and sufficient condition that also considers aggregations and operators.

An interesting related system is Orchid [23]. It adopts an abstract data model (*OHM*) for ETL flows, whose instances are then translated first into schema mappings and then into executable ETL scripts. Moreover, this system handles the generated mappings for some kind of multi-query optimization, providing a composition algorithm, which, however, does not handle aggregations nor consider mathematical operations. We observe that, instead, aggregations and calculations in general are very frequent in ETL and fundamental in statistical data processing.

Two main architectural approaches are present for the processing of large amounts of statistical data: scaling the statistical tool or scaling the data management system [39]. Systems in the first category tend to stick to a specific tool and language and provide a scalable infrastructure for it. Many examples are present and can be classified according to the level of abstraction of the interface they propose: *message-passing* techniques deliver low level APIs, while *parallel computing* shift towards fewer highly parallelizable high-level primitives [34]. Systems in the second category try to incorporate analytics into the data management system and rely on its scalability. Cohen et al. [19] present one of the first proposals to extend SQL with statistical functions and implement them in a relational DBMS. Away from the relational model, many other examples, such as Mahout,[13] *SystemML* [12] or Spark,[14] directly implement many algorithms on top of Hadoop and basically provide an high-level abstraction over it. Finally, others [40] foster a radically diverging approach, attempting to give a completely new data model for scientific data processing and designing ad hoc systems.

EXLEngine attempts a meet-in-the-middle approach, where the specific capabilities of statistical tools are recognized with the driving principle of "not reinventing the wheel". At the same time, we observe that RDBMS and data management systems in general (including ETL tools) are still a primary choice in data processing. Therefore we decided to stick neither to a language nor to a system and provide a language- and system-independent approach. For this, the use of schema mappings is essential and allows for a complete decoupling, from both a syntactical/semantic and a performance point of view.

In this respect, a system that is somehow close to EXLEngine is *Ricardo* [20]. Unlike EXLEngine, they provide an R-like interface on top of Hadoop. While Ricardo allows for the specification of sophisticated procedural programs, where coding skills of the users are valuable, our system offers a simpler declarative interface and is even more agnostic on the language and on the system side. Indeed, we rely even more heavily on optimization capabilities of data management systems and handle a variety of them, including (but not only) R. In this sense, EXLEngine could even exploit Ricardo as a target system.

[13] http://mahout.apache.org
[14] http://spark.apache.org

9 Conclusions

Statistical data processing is a core task for many companies and organizations and it is acquiring more and more relevance in the so-called Big Data settings and with the recent renewed interest in machine learning. In this work, we contribute in bridging the gap between the high-level specification of the statistical programs needed to achieve the desired tasks and the underlying technical implementations.

One of the major issues in organizations is the adoption of a range of different tools, each with its specific language and features. Statisticians and analysts would like to specify the programs at high level, independently of the underlying execution platform, yet leveraging at the same time the whole range of possible execution environments. Here we consider the framework of schema mappings and show that st-tgds, non-recursive t-tgds and egds capture a useful fragment for the description of statistical programs, in which the chase procedure for data exchange always terminates and can be executed in a scalable way. We present EXLEngine, a real execution environment adopted by Bank of Italy, which embodies many techniques from our approach, including advanced optimizations based on tgd composition.

In this paper, we touch on a complexity characterization of the fragment, which testifies the possibility of parallel implementations, essential for scalability. Although our execution optimizer is able to split the calculation tasks and assign them to different engines, which can run in parallel, within the single engine, the parallelization choices are completely delegated to the back-end. In our future work we plan to refine the interaction with back-ends in order to guide and tune their execution plans at best.

Also, the approach still has some limitations that are being addressed. Some operators are extremely natural to be supported by certain target systems, as the respective language directly incorporates executable versions for them. It is the case with more advanced statistical transformations and specialized tools. Other cases are more difficult, since one single high-level statistical operator may correspond to several operations in the back-end. We are aiming at a complete support, which would include the translation of almost any operator to every target system.

References

1. M. Arenas, P. Barceló, L. Libkin, F. Murlak. Foundations of Data Exchange. *Cambridge University Press*, 2014.
2. M. Arenas, R. Fagin, and A. Nash. Composition with target constraints. *Logical Methods in Computer Science*, 7(3), 2011.
3. M. Arenas, G. Gottlob, and A. Pieris. Expressive languages for querying the semantic web. In *PODS*, pages 14–26, 2014.
4. P. Atzeni, L. Bellomarini, and F. Bugiotti. Exlengine: executable schema mappings for statistical data processing. In *EDBT*, pages 672–682, 2013.
5. 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.

6. 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.
7. P. Atzeni, P. Cappellari, R. Torlone, P. Bernstein, and G. Gianforme. Model-independent schema translation. *VLDB Journal*, 17:1347–1370, 2008.
8. Z. Bellahsene, A. Bonifati, E. Rahm. Schema Matching and Mapping. Data-Centric Systems and Applications. *Springer*. 2011.
9. L. Bellomarini, G. Gottlob, A. Pieris, and E. Sallinger. Swift logic for big data and knowledge graphs. In *IJCAI*, pages 2–10, 2017.
10. M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, E. Tsamoura. Benchmarking the Chase. In PODS, pages 37–52, 2017.
11. P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD Conference*, pages 1–12, 2007.
12. M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in systemml. *PVLDB*, 7(7):553–564, 2014.
13. A. Bonifati, E. Q. Chang, T. Ho, L. V. S. Lakshmanan, and R. Pottinger. Heptox: Marrying XML and heterogeneity in your P2P databases. In *VLDB*, pages 1267–1270, 2005.
14. P. J. Brockwell and R. A. Davis, editors. *Introduction to Time Series and Forecasting*. Springer, 2002.
15. A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *PODS*, pages 77–86, 2009.
16. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Data complexity of query answering in description logics (extended abstract). In *IJCAI*, pages 4163–4167, 2015.
17. S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, PODS '98, pages 34–43, New York, NY, USA, 1998. ACM.
18. S. Chaudhuri and K. Shim. Including group-by in query optimization. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB*, pages 354–366. Morgan Kaufmann, 1994.
19. J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009.
20. S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: integrating R and hadoop. In *SIGMOD*, pages 987–998, 2010.
21. V. Del Vecchio. Statistical data and concepts representation. *Bank of Italy*, 1997. `http://goo.gl/YIAqDp`.
22. V. Del Vecchio, F. Di Giovanni, and S. Pambianco. The "matrix" model. *Bank of Italy*, 2007. `http://goo.gl/Dj2XTO`.
23. S. Dessloch, M. Hernández, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: Integrating schema mapping and etl. In *ICDE*, pages 1307–1316, 2008.
24. F. Di Giovanni and D. Piazza. Processing and managing statistical data: a national central bank experience. *Bank of Italy*, 2009. `http://goo.gl/ZNi5zh`.
25. 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.
26. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *In ICDT*, pages 207–224, 2003.
27. R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.
28. R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.
29. R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Schema mapping evolution through composition and inversion. In *Schema Matching and Mapping*, pages 191–222. 2011.
30. G. Gottlob, R. Pichler, and V. Savenkov. Normalization and optimization of schema mappings. *PVLDB*, 2(1):1102–1113, 2009.
31. L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, pages 805–810. ACM, 2005.

32. P. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75, 2005.
33. P. G. Kolaitis, J. Panttaja, and W. C. Tan. The complexity of data exchange. In *SIGMOD*, pages 30–39, 2006.
34. E. Mahdi. A survey of r software for parallel computing. *American Journal of Applied Mathematics and Statistics*, 2(4):224–230, 2014.
35. G. Mecca, P. Papotti, and S. Raunich. Core schema mappings: Scalable core computations in data exchange. In *Inf. Syst.*, pages 37(7):677–711, 2012.
36. I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, pages 264–277, 1990.
37. J. O. Ramsay, G. Hooker, and S. Graves, editors. *Functional Data Analysis with R and Matlab*. Springer, 2009.
38. E. Sallinger. Reasoning about schema mappings. In *Data Exchange, Integration, and Streams*, pages 97–127. 2013.
39. M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. State of the art in parallel computing with r. *Journal of Statistical Software*, 31(1):1–27, 8 2009.
40. M. Stonebraker, J. Becla, D. J. DeWitt, K. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and scidb. In *CIDR*, 2009.