Smart Data Exchange
(DISCUSSION PAPER)

Sergio Greco[1], Michele Ianni[1], Elio Masciari[2],
Domenico Saccà[1], and Irina Trubitsyna[1]
elio.masciari@unina.it
{greco, ianni, sacca, trubitsyna}@dimes.unical.it

[1] DIMES-Università della Calabria, 87036 Rende (CS), Italy
[2] DIETI-Università degli Studi di Napoli Federico II, 80125 Napoli (NA), Italy

Abstract. The problem of exchanging data, even considering incomplete and heterogeneous data, has been deeply investigated in the last years. The approaches proposed so far are quite rigid as they refer to fixed schema and/or are based on a deductive approach consisting in the use of a fixed set of (mapping) rules. In this paper we describe a smart data exchange framework integrating deductive and inductive techniques to obtain new knowledge. The use of graph-based representation of source and target data, together with the midway relational database and the extraction of new knowledge allow us to manage dynamic databases where also features of data may change over the time.

1 Introduction

Many proposal have been made for data exchange among heterogeneous sources. However, for the best of our knowledge, no generalization of the consolidated data exchange framework, that supports both the extraction of new knowledge and flexible representation of heterogeneous data has been defined so far. In this paper we describe an extension of the data exchange framework, called *Smart Data Exchange (SDE)*, recently proposed in [13], that addresses these issues. The global scenario is showed in Fig. 1, where (i) *Source*, *Target* and *Midway* are three databases, (ii) $\Sigma_{\mathtt{SM}}$ and $\Sigma_{\mathtt{MT}}$ are extended TGDs (Tuple Generating Dependencies) mapping data from *Source* to *Midway* and standard TGDs mapping data from *Midway* to *Target*, respectively, (iii) $\Sigma_{\mathtt{M}}$ and $\Sigma_{\mathtt{T}}$ are extended TGDs defined over the *Midway* database and standard TGDs and EGDs (Equality Generating Dependencies) defined over the *Target* database, respectively.

The idea is that, by allowing an intermediate database and a richer language to derive new information, as well as information obtained by analyzing data, we may define more powerful and flexible tools for data exchange. To have a flexible and general representation of source and target databases, following the RDF approach, we decided to model them using a graph-based formalism where data are stored into ternary relations. Differently from other formalisms where data are stored into a unique relation [3, 18], we consider multiple ternary relations, but analogously to RDF our graph data consist

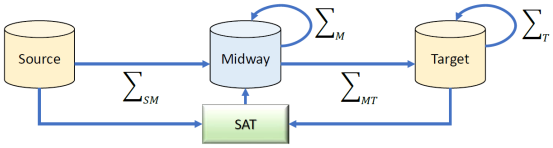

Fig. 1. Architecture

of tuples of the form (i, a, v) with i being a (resource) identifier, a an attribute name and v a value (either a constant or an identifier). The midway database is a relational database containing tuples of any arity that are either imported from the source database using the $\Sigma_{\mathtt{SM}}$ source-midway mapping rules, or are generated by the *Smart Analyzer Tool (SAT)*. Data dependencies denoted by $\Sigma_{\mathtt{M}}$ are used to generate new information useful to enrich the target database. The target database is graph-based and is built by importing data from the midway database using the mapping rules $\Sigma_{\mathtt{MT}}$. Finally, as in the standard data exchange scenario, $\Sigma_{\mathtt{T}}$ consists of a set of TGDs or EGDs.

While we assume that *SAT* is a black-box aiming to produce new data stored in the midway database (in our prototype it implements several data mining algorithms specific for the domain application), the language used to define $\Sigma_{\mathtt{M}}$ is a rich logical language that makes use of comparison predicates, *aggregate predicates* and nondeterministic *choice* constructs (well studied in the past by the community of logic databases). This language allows to obtain more realistic information that could be used in a flexible way by both data experts (for analysis purposes) and inexperienced users (for a better navigation through data). Another peculiarity of our framework is that the data exchange takes also advantages of information (under the form of facts) generated by a data analyzer module and stored into the midway database, together with the data extracted from the source database.

The idea to model data analysis during the data mapping is present in Data Posting framework [8] and its simplified version [19]. Differently from SDE, the framework proposed in [8, 19] does not use graph-based formalisms for source and target data and does not have an intermediate level for data analysis. Moreover, it does not admit the use of existentially quantified variables in the mapping rules and uncertain values in the body of mapping rules can be represented only by means of *non-deterministic* variables, whose values can be selected from specific relations, possibly restricted by target *count constraints*.

2 Background

A *Tuple Generating Dependency (TGD)* is formula of the form: $\forall \mathbf{x}\, \phi(\mathbf{x}) \rightarrow \exists \mathbf{y}\, \psi(\mathbf{x}, \mathbf{y})$, where $\phi(\mathbf{x})$ and $\psi(\mathbf{x}, \mathbf{y})$ are conjunctions of atoms, and $\mathbf{x}, \mathbf{y}$ are lists of variables. *Full TGDs* are TGDs without existentially quantified variables. An *Equality Generating Dependency (EGD)* is a formula of the form: $\forall \mathbf{x}\, \phi(\mathbf{x}) \rightarrow x_1 = x_2$, where $\phi(\mathbf{x})$ is conjunction of atoms, while x_1 and x_2 are variables in $\mathbf{x}$. In our formulae we omit the universal quantifiers, when their presence is clear from the context.

Let $S = S_1, ..., S_n$ and $T = T_1, ..., T_m$ be two disjoint schemas. We refer to S (resp. T) as the source (resp. target) schema and to the S_i's (resp. T_j's) as the source (resp.

target) relation symbols. Instances over S (resp. T) will be called source (resp. target) instances. If I is a source instance and J is a target instance, then we write $\langle I, J\rangle$ for the instance K over the schema $S \cup T$ such that $K(S_i) = I(S_i)$ and $K(T_j) = J(T_j)$, for $i \leq n$ and $j \leq m$.

The data exchange setting [11, 2] is a tuple $(S, T, \Sigma_{st}, \Sigma_t)$, where S is the source schema, T is the target schema, Σ_t are dependencies over T and Σ_{st} are source-to-target TGDs. The dependencies in Σ_{st} map data from the source to the target schema and are TGDs of the form $\forall \mathbf{x}(\, \phi_s(\mathbf{x}) \;\rightarrow\; \exists \mathbf{y}\;\; \psi_t(\mathbf{x}, \mathbf{y})\,)$, where $\phi_s(\mathbf{x})$ and $\psi_t(\mathbf{x}, \mathbf{y})$ are conjunctions of atomic formulas on S and T, respectively. Dependencies in Σ_{st} are also called mapping dependencies. Dependencies in Σ_t specify constraints on the target schema and can be either TGDs or EGDs.

The data exchange problem associated with this setting is the following: given a finite source instance I, find a finite target instance J such that $\langle I, J\rangle$ satisfies Σ_{st} and J satisfies Σ_t. Such a J is called a solution for I. The computation of an universal solution (the compact representation of all possible solutions) can be done by means of the fixpoint chase algorithm, when it terminates [9]. Decidable (sufficient) conditions ensuring chase termination can be found in [16, 6, 15].

Queries over relational database can be expressed using Relational Algebra, or alternative equivalent languages such safe Relational Calculus and safe, nonrecursive Datalog (with negation). To make query languages more expressive, several additional feature have been added to these languages including the possibility to manage bags (SQL), aggregates (SQL), recursion (Datalog), existential variables (Datalog$^{\pm}$) and others.

The choice constructs [20, 17] have been introduced in Datalog to get an increase in expressive power and to obtain simple declarative formulations of classical combinatorial problems, such as those which can be solved by means of greedy algorithms [20].

A *choice atom* is of the form $choice((\mathbf{x}), (\mathbf{y}))$, where $\mathbf{x}$ and $\mathbf{y}$ are lists of variables such that $\mathbf{x} \cap \mathbf{y} = \emptyset$, and can occur in the body of a rule. Its intuitive meaning is to force each initialization of $\mathbf{x}$ to be associated with a unique initialization of $\mathbf{y}$, thus making the result of executing of a corresponding rule nondeterministic. A *choice rule* is of the form: $A(\mathbf{w}) \leftarrow B(\mathbf{z}), choice((\mathbf{x}), (\mathbf{y}))$, where $A(\mathbf{w})$ is an atom, $B(\mathbf{z})$ is a conjunction of atoms, $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ and $\mathbf{w}$ are lists of variables such that $\mathbf{x}, \mathbf{y}, \mathbf{w} \subseteq \mathbf{z}$. The atom $choice((\mathbf{x}), (\mathbf{y}))$ is used to enforce the *functional dependency* $\mathbf{x} \rightarrow \mathbf{y}$ on the set of atoms derived by means of the rule.

The *choice-least* and *choice-most* constructs [17] specialize the choice construct so as to force greedy selections among alternative choices—these turn out to be particularly useful to express classical greedy algorithms. A *choice-least* (resp. *choice-most*) *atom* is of the form $choice\_least((\mathbf{x}), (c))$ (resp. $choice\_most((\mathbf{x}), (c))$), where $\mathbf{x}$ is a list of variables and c is a single variable ranging over an ordered domain. A $choice\_least((\mathbf{x}), (c))$ (resp. $choice\_most((\mathbf{x}), (c))$) atom in a rule indicates that the functional dependency defined by the atom $choice((\mathbf{x}), (c))$ is to be satisfied, and the c value assigned to a certain value of $\mathbf{x}$ has to be the minimum (resp. maximum) one among the candidate values. The body of a rule may contain even more than one choice constructs, but only one choice-least or choice-most atom. For instance, the rule $\mathtt{p}(\mathtt{x}, \mathtt{y}, \mathtt{c}) \leftarrow \mathtt{q}(\mathtt{x}, \mathtt{y}, \mathtt{c}), \mathtt{choice}((\mathtt{x}), (\mathtt{y})), \mathtt{choice\_least}((\mathtt{x}), (\mathtt{c}))$ imposes the functional dependency $\mathtt{x} \rightarrow \mathtt{y}, \mathtt{c}$ on the possible instances of $\mathtt{p}$. In addition, for each value of

x, the minimum among the candidate values of c must be chosen. For instance, assuming that q is defined by the facts $\mathtt{q}(\mathtt{a},\mathtt{b},1)$ and $\mathtt{q}(\mathtt{a},\mathtt{d},2)$, from the rule above we might derive either $\mathtt{p}(\mathtt{a},\mathtt{b},1)$ or $\mathtt{p}(\mathtt{a},\mathtt{d},2)$. However, the choice-least atom introduces the additional requirement that the minimum value on the third attribute has to be chosen, so that only $\mathtt{p}(\mathtt{a},\mathtt{b},1)$ is derived. The formal semantics is defined by rewriting rules with choice atoms into rules with negated literals and selecting (nondeterministically) one of the stable models of the rewritten program [20, 17].

3 Smart Data Exchange

In this section we present the data exchange framework informally discussed in the Introduction. We assume the existence of the following countably infinite sets: *relation names* $\mathcal{R}$, *identifiers* $\mathcal{I}$, *attribute names* $\mathcal{A}$, *constants* $\mathcal{C}$, *nulls* $\mathcal{N}$ and *variables* $\mathcal{V}$. The set of relation symbols (also called predicate symbols) is partitioned into three countable sets denoted by $\mathcal{R}_\mathtt{S}$ (source relations), $\mathcal{R}_\mathtt{T}$ (target relations) and $\mathcal{R}_\mathtt{M}$ (midway relations), whereas $\mathcal{D}_\mathtt{S} = \mathcal{I} \cup \mathcal{A} \cup \mathcal{C}$, $\mathcal{D}_\mathtt{T} = \mathcal{I} \cup \mathcal{A} \cup \mathcal{C} \cup \mathcal{N}$ and $\mathcal{D}_\mathtt{M} = \mathcal{I} \cup \mathcal{A} \cup \mathcal{C}$ denote the domains of relations $\mathcal{R}_\mathtt{S}$, $\mathcal{R}_\mathtt{T}$ and $\mathcal{R}_\mathtt{M}$, respectively. Relations in $\mathcal{R}_\mathtt{S}$ and $\mathcal{R}_\mathtt{T}$ have arity 3 and take values from $\mathcal{I} \times \mathcal{A} \times \mathcal{I} \cup \mathcal{C}$ and $\mathcal{I} \cup \mathcal{N} \times \mathcal{A} \cup \mathcal{N} \times \mathcal{I} \cup \mathcal{C} \cup \mathcal{N}$ respectively, whereas relations in $\mathcal{R}_\mathtt{M}$ may have any arity n and take values from $\mathcal{D}_\mathtt{M}^n$. The main difference between the source and the target databases is that the target database may also have nulls (corresponding to blank nodes in RDF) which are introduced to satisfy constraints. The set of source (resp. target, midway) relations define the source (resp. target, midway) database whose schema is denoted by S (resp. T, M).

The model defined above states that the source and target databases are graph-based databases stored in a relational database (using triples), whereas the midway database is a standard relational database. This choice is due to the fact that we would exchange data among heterogeneous databases and we want model data whose schema may change over the time.

Extended TGDs. An atom is of the form $p(t_1, ..., t_n)$, where $t_1, ..., t_n$ are terms (standard atom), or of the form $t_1 \theta t_2$, where θ is a comparison predicates and t_1, t_2 are terms (built-in atom). A literal is an atom A (positive literal) or its negation $\neg A$ (negative literal). A conjunction of literals is of the form $B_1 \wedge \cdots \wedge B_n$ where $B_1, ..., B_n$ are literals. A conjunction of literals is said to be *safe* if all variables occurring in built-in atoms and negated literals also appear in positive literals. From now on we assume that whenever we consider conjunctions of literals they are safe.

Definition 1. An extended TGD (ETGD) is a universally quantified implication formula of one of the following forms:

- $\varphi(x) \wedge choice(x') \rightarrow \psi(w)$,
  where $\varphi(x)$ is a safe conjunction of standard and built-in literals, $\psi(w)$ is a conjunction of atoms, $w \subseteq x$ and $choice(x')$, with $x' \subseteq x$, is a possibly empty conjunction of $choice$-atoms;
- $\varphi(x) \rightarrow q(w_0, c_1\langle w_1 \rangle, ..., c_n\langle w_n \rangle)$ (called aggregate dependency)
  where $\varphi(x)$ is a safe conjunction of standard and built-in literals, $c_1, ..., c_n$ denote aggregate functions (e.g., $min, max, sum, count$), w_0 (called "group-by" variables) and $w_1, ..., w_n$ are lists of variables such that $w_0, ..., w_n \subseteq x$ and $w_0 \cap w_i = \emptyset$ $\forall i \in [1, n]$. □

For any set Σ of data dependencies, the *dependency graph* $G_\Sigma = (V, E)$ is built as follows: V consists of the predicate symbols occurring in Σ, whereas there is an edge from p to q if there is a dependency having p in the body and q in the head. Moreover, the edge is labeled with $\neg$ (resp. ag) if p occurs negated (resp. occurs in the head atom which contains aggregate functions). A set of dependency is said to be *stratified* if the depencecy graph does not contain cycles with labeled edges. Observe that since standard dependencies are positive (that is, all body literals are positive), to check stratification it is sufficient to check stratification of ETGDs. From now on we assume that our dependencies are stratified.

The semantics of ETGDs with choice atoms can be defined as in the case of Datalog rules. To this end, in order to eliminate head conjunctions, any ETGDs $r : \varphi(x) \rightarrow p_1(y_1) \wedge \cdots \wedge p_k(y_k)$ having $k > 1$ atoms in the head is rewritten into: (i) k ETGDs $r_i : \varphi(x) \rightarrow p_i(y_i)$ ($i \in [1, k]$), if r does not contain choice atoms, and (ii) $k+1$ ETGDs $r_0, ..., r_k$ if r contains choice atoms, where $r_0 = \varphi(x) \rightarrow h_r(x)$ and $r_i = h_r(x) \rightarrow p_i(y_i)$ (with $i \in 1, k]$), where h_r is a fresh new predicate. Regarding the semantics of ETGDs with aggregate functions, as the set of ETGDs is stratified, we can partition it into strata so that, if a stratum contains an ETGD with aggregate functions, then it does not contain any other ETGD, and compute one stratum at time following topological order defined over the strata by the dependency graph. The computation of an ETGD with aggregate functions can be carried out in the same way of computing SQL queries with aggregates, as body predicates have been already computed and, therefore, they correspond to database relations in SQL (see also [12]).

Smart data exchange. Now we formally define the Smart Data Exchange Framework.

Definition 2. A *Smart Data Exchange (SDE)* is a tuple $(S, M, T, \Sigma_{\mathtt{SM}}, \Sigma_{\mathtt{MT}}, \Sigma_{\mathtt{M}}, \Sigma_{\mathtt{T}})$, where:

- S is the source schema containing predicates taken from $\mathcal{R}_{\mathtt{S}}$,
- M is the midway schema containing predicates taken from $\mathcal{R}_{\mathtt{M}}$,
- T is the target schema containing predicates taken from $\mathcal{R}_{\mathtt{T}}$,
- $\Sigma_{\mathtt{SM}}$ is a source-to-midway set of safe ETGDs,
- $\Sigma_{\mathtt{M}}$ is a midway-to-midway set of safe, stratified ETGDs,
- $\Sigma_{\mathtt{MT}}$ is a midway-to-target set of standard TGDs, and
- $\Sigma_{\mathtt{T}}$ is a target-to-target set of standard TGDs and standard EGDs. □

As already pointed out, we assume that the source and target relations are graph-based data and every class of objects is modeled as a named set of triples. RDF graph, microdata and JSON representations are very closed to this description [1]. The midway database is a relational database containing relations of any arity including data in the format of the source and target database.

For any SDE $E = (S, M, T, \Sigma_{\mathtt{SM}}, \Sigma_{\mathtt{M}}, \Sigma_{\mathtt{MT}}, \Sigma_{\mathtt{T}})$, we shall use the following notation: $\Sigma_1 = \Sigma_{\mathtt{SM}} \cup \Sigma_{\mathtt{M}}$, $\Sigma_2 = \Sigma_{\mathtt{MT}} \cup \Sigma_{\mathtt{T}}$, and $\Sigma = \Sigma_1 \cup \Sigma_2$.

Example 1. Consider a graph relation in the source database containing facts of the form $\mathtt{graph}(\mathtt{id}_1, \mathtt{edge}, \mathtt{id}_2)$ where $\mathtt{id}_1$ and $\mathtt{id}_2$ are node identifiers and edge is an attribute value denoting that there is an edge from $\mathtt{id}_1$ to $\mathtt{id}_2$. The data exchange problem consists in extracting a spanning tree to be stored in the target database. From the source database we can import edges and nodes in the midway database, as defined in the set

$\Sigma_{\mathtt{SM}}$ consisting of the ETGD: $\mathtt{graph}(\mathtt{x}, \mathtt{edge}, \mathtt{y}) \rightarrow \mathtt{edge}(\mathtt{x}, \mathtt{y}) \wedge \mathtt{node}(\mathtt{x}) \wedge \mathtt{node}(\mathtt{y})$. Assume now that the midway database also contains a fact $\mathtt{root}(0)$, for instance generated by the SAT module or imported from the source database using another source-to-midway dependency. The next set of ETGDs $\Sigma_{\mathtt{M}}$ shows how it is possible to generate a spanning tree rooted in the node x denoted by fact $\mathtt{root}(\mathtt{x})$.

$$\mathtt{root}(\mathtt{x}) \rightarrow \mathtt{st}(\mathtt{nil}, \mathtt{x})$$
$$\mathtt{st}(\mathtt{z}, \mathtt{x}) \wedge \mathtt{edge}(\mathtt{x}, \mathtt{y}), \mathtt{choice}((\mathtt{y}), (\mathtt{x})), \rightarrow \mathtt{st}(\mathtt{x}, \mathtt{y}) \wedge \mathtt{connected}(\mathtt{y})$$
$$\mathtt{node}(\mathtt{x}) \wedge \neg\mathtt{connected}(\mathtt{x}), \mathtt{choice}((), (\mathtt{x})) \rightarrow \mathtt{nextRoot}(\mathtt{x})$$

Here the first ETGD is used to start the computation by deriving an edge ending in the root node (the starting node is the dummy node nil), whereas the second ETGD, imposing the functional dependency $\mathtt{y} \rightarrow \mathtt{x}$, guarantees that the set of selected nodes is a spanning tree. The last ETGD in $\Sigma_{\mathtt{M}}$ gives a node not belonging to the spanning tree if the graph is not connected; this node can be used in the future as a root to compute another spanning tree.

Spanning tree edges and nextRoot facts are then imported in the target database (as triples) using the below set of TGDs $\Sigma_{\mathtt{MT}}$:

$$\mathtt{st}(\mathtt{x}, \mathtt{y}) \rightarrow \mathtt{graph}(\mathtt{x}, \mathtt{st}, \mathtt{y})$$
$$\mathtt{nextRoot}(\mathtt{x}) \rightarrow \exists\mathtt{y}\ \mathtt{graph}(\mathtt{nil}, \mathtt{root}, \mathtt{x})$$

Information stored in the target database are next analyzed to generate new information which will be stored in the midway database. For instance, from a fact $\mathtt{graph}(\mathtt{nil}, \mathtt{root}, \mathtt{x})$ the SAT module could generate a fact $\mathtt{root}(\mathtt{x})$ which, after been stored in the midway database, could generate the computation of another spanning tree rooted in x. □

Note that the process described in the previous example and showed in Fig. 1 is supervised, that is the activation of the module SAT is performed by the user.

The smart data exchange problem associated with this setting is the following: given a finite source instance I over a schema S and a smart data exchange $E = (S, M, T, \Sigma_{\mathtt{SM}}, \Sigma_{\mathtt{M}}, \Sigma_{\mathtt{MT}}, \Sigma_{\mathtt{T}})$, find finite instances J and K over the schemas M and T, respectively, such that $\langle I, J\rangle$ satisfies $\Sigma_{\mathtt{SM}}$, J satisfies $\Sigma_{\mathtt{M}}$, $\langle J, K\rangle$ satisfies $\Sigma_{\mathtt{MT}}$ and K satisfies $\Sigma_{\mathtt{M}}$. The pair $\langle J, K\rangle$ is called a solution (or model) for $\langle I, E\rangle$, or equivalently for $\langle I, \Sigma\rangle$, where let $\Sigma_1 = \Sigma_{\mathtt{SM}} \cup \Sigma_{\mathtt{M}}$ and $\Sigma_2 = \Sigma_{\mathtt{MT}} \cup \Sigma_{\mathtt{T}}$, $\Sigma = \Sigma_1 \cup \Sigma_2$. The set of solutions for $\langle I, E\rangle$ (or equivalently for $\langle I, \Sigma\rangle$) is denoted by $Sol(I, E)$, (resp. $Sol(I, \Sigma)$). Moreover, J is also called solution (or model) for $\langle I, \Sigma_1\rangle$ and, in cascade, K is a solution (or model) for $\langle J, \Sigma_2\rangle$. Thus, the problem of finding solutions can be split into two problems: (i) finding a solutions J for I, and (ii) finding solutions K for J. The set of solutions for $\langle I, \Sigma_1\rangle$ is denoted by $Sol(I, \Sigma_1)$, whereas the set of solutions for $\langle J, \Sigma_2\rangle$ is $Sol(J, \Sigma_2)$. Therefore, given a smart data exchange framework, starting from I with dependency Σ_1 we find solutions $J_1, ..., J_m$ and, then starting from every J_i $(1 \leq i \leq m)$ with dependency Σ_2 we find solutions $K_{i_1}, ..., K_{i_n}$. Observe that m is always finite, whereas i_n, in the general case, is not guaranteed to be finite [10].

Query answering. Since we have multiple models, we distinguish the *certain answer* derived from all models and the *nondeterministic answer*.

Definition 3. *Given a query Q, a database I and an SDE E, the certain answer of Q over $\langle I, E\rangle$ is* $Certain(Q, I, E) = \bigcap_{\langle J,K\rangle \in Sol(I,E)} Q(K)$. □

Since the certain answer could be equivalently defined as $Certain(Q, I, E) = \bigcap_{J\in Sol(I,\Sigma_1)\wedge K\in Sol(J,\Sigma_2)} Q(K)$, its computation can be optimized by considering the four sets of dependencies separately, that is by first computing solutions J' for $\langle I, \Sigma_{\mathtt{ST}}\rangle$, then solutions J for $\langle J', \Sigma_{\mathtt{M}}\rangle$, next solutions K' for $\langle J, \Sigma_{\mathtt{MT}}\rangle$, and, finally solutions K for $\langle K', \Sigma_{\mathtt{T}}\rangle$. Let us now introduce the concepts of homomorphism and universal model.

A *homomorphism* from a set of atoms A_1 to a set of atoms A_2 is a mapping h from the domain of A_1 (set of terms occurring in A_1) to the domain of A_2 such that: (i) $h(c) = c$, for every $c \in \mathsf{Const}(A_1)$; and (ii) for every atom $R(t_1, \ldots, t_n)$ in A_1, we have that $R(h(t_1), \ldots, h(t_n))$ is in A_2. With a slight abuse of notation, we apply h also to sets of atoms and thus, for a given set of atoms A, we define $h(A) = \{R(h(t_1), \ldots, h(t_n)) \mid R(t_1, \ldots, t_n) \in A\}$.

Definition 4. A *universal model* of (I, Σ) is a model $\langle J, K\rangle$ of (I, Σ) such that for every model $\langle J', K'\rangle$ of (I, Σ) there exists a homomorphism from K to K'. The set of all universal solutions of (I, Σ) will be denoted by $USol(I, \Sigma)$. □

Proposition 1. *For any positive query Q, database I and SDE E, the certain answer of Q over $\langle I, E\rangle$ is* $Certain(Q, I, E) = \bigcap_{J \in Sol(I,\Sigma_1)} Q(K_J)_{\downarrow}$, *where K_J is any universal solution of $\langle J, \Sigma_2\rangle$ and $Q(K_J)_{\downarrow}$ is the result of computing naively (i.e. considering nulls as constants) $Q(K_J)$ and deleting tuples with nulls.* □

Universal solutions for $\langle J, \Sigma_2\rangle$ can be easily computed by applying the classical fixpoint algorithm called *Chase* which computes a subset of universal solutions called *canonical* [4, 10].

As previously discussed, in several cases we are not interested in all models of $\langle I, \Sigma_1\rangle$, but only in one selected nondeterministically (e.g. the set of edges of any minimum spanning tree). Thus, we now introduce the definition of *nondeterminitic answer*.

Definition 5. The *nondeterministic answer* to a query Q over a database instance I and SDE $E = (S, M, T, \Sigma_{\mathtt{SM}}, \Sigma_{\mathtt{M}}, \Sigma_{\mathtt{MT}}, \Sigma_{\mathtt{T}})$ is $NonDet(Q, I, E) = \bigcap_{K \in Sol(J,\Sigma_2)} Q(K)$, where, J is a model for $\langle I, \Sigma_1\rangle$ selected nondeterministically. □

Observe that the nondeterministic choice of the model is applied only to dependencies in Σ_1, where users express explicitly that they want select nondeterministically a subset of tuples, whereas for $\langle J, \Sigma_2\rangle$ we consider all models. Note that, two evaluations of the nondeterministic answers could give different answers (as the choices made could be different) and that the responsibility of computing nondeterministic answers, instead of certain answers, is left to the user. Indeed, in several cases the user is not interested in specific models, but only in one model satisfying some properties (e.g. any spanning tree). Therefore, we introduce the concept of universal nondeterministc model.

Definition 6. A *universal nondeterministic model* for $\langle I, \Sigma\rangle$ is any universal model in $USol(D, \Sigma)$ selected nondeterministically. □

Proposition 2. *For any positive query Q, database I and SDE $E = (S, M, T, \Sigma_{\mathtt{SM}}, \Sigma_{\mathtt{M}}, \Sigma_{\mathtt{MT}}, \Sigma_{\mathtt{T}})$, the nondeterministic answer of Q over $\langle I, E\rangle$ is* $NonDet(Q, I, E) = Q(K_J)_{\downarrow}$, *where, let J be any model for $\langle I, \Sigma_1\rangle$ selected nondeterministically, K_J is a universal solution of $\langle J, \Sigma_2\rangle$* □

4 Conclusion

In this paper we have discussed a framework supporting both analysis and flexible representation of heterogeneous data. The use of graph-based representation of source and target data, together with the extraction of new knowledge made by the SAT tool allow us to manage dynamic databases where also features of data may change over the time. The theoretical complexity of the approach is discussed in [13]. Future work should include the use of more powerful declarative languages for the midway, where limited use of function symbols is allowed [5, 7, 14].

References

1. R. Angles and C. Gutierrez. An introduction to graph data management. In *Graph Data Management, Fundamental Issues and Recent Developments.*, pages 1–32. 2018.
2. M. Arenas, P. Barceló, R. Fagin, and L. Libkin. Locally consistent transformations and query answering in data exchange. In *PODS*, 2004.
3. M. Arenas, G. Gottlob, and A. Pieris. Expressive languages for querying the semantic web. *ACM Trans. Database Syst.*, 43(3):13:1–13:45, 2018.
4. C. Beeri and M. Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984.
5. M. Calautti, S. Greco, C. Molinaro, and I. Trubitsyna. Logic program termination analysis using atom sizes. In *IJCAI*, pages 2833–2839. AAAI Press, 2015.
6. M. Calautti, S. Greco, C. Molinaro, and I. Trubitsyna. Exploiting equality generating dependencies in checking chase termination. *PVLDB*, 9(5):396–407, 2016.
7. M. Calautti, S. Greco, F. Spezzano, and I. Trubitsyna. Checking termination of bottom-up evaluation of logic programs with function symbols. *TPLP*, 15(6):854–889, 2015.
8. N. Cassavia, E. Masciari, C. Pulice, and D. Saccà. Discovering user behavioral features to enhance information search on big data. *TiiS*, 7(2):7:1–7:33, 2017.
9. A. Deutsch, A. Nash, and J. B. Remmel. The chase revisited. In *PODS*, 2008.
10. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.
11. R. Fagin, P. G. Kolaitis, and L. Popa. Data Exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.
12. S. Greco. Dynamic programming in datalog with aggregates. *IEEE Trans. Knowl. Data Eng.*, 11(2):265–283, 1999.
13. S. Greco, E. Masciari, D. Saccà, and I. Trubitsyna. HIKE: A step beyond data exchange. In *ER Conference*, 2019.
14. S. Greco, C. Molinaro, and I. Trubitsyna. Logic programming with function symbols: Checking termination of bottom-up evaluation through program adornments. *Theory Pract. Log. Program.*, 13(4-5):737–752, 2013.
15. S. Greco, F. Spezzano, and I. Trubitsyna. Stratification criteria and rewriting techniques for checking chase termination. *PVLDB*, 4(11):1158–1168, 2011.
16. S. Greco, F. Spezzano, and I. Trubitsyna. Checking chase termination: Cyclicity analysis and rewriting techniques. *IEEE Trans. Knowl. Data Eng.*, 27(3):621–635, 2015.
17. S. Greco and C. Zaniolo. Greedy algorithms in datalog. *TPLP*, 1(4):381–407, 2001.
18. L. Libkin, J. L. Reutter, A. Soto, and D. Vrgoc. Trial: A navigational algebra for RDF triplestores. *ACM Trans. Database Syst.*, 43(1):5:1–5:46, 2018.
19. E. Masciari, I. Trubitsyna, and D. Saccà. Simplified data posting in practice. In *IDEAS*, pages 29:1–29:7, 2019.
20. D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proc. PODS Conf.*, pages 205–217, 1990.