Query Processing in Data Integration

Paolo Guagliardo[1] and Piotr Wieczorek[2]

1 **KRDB Research Centre, Free University of Bozen-Bolzano, Italy**
guagliardo@inf.unibz.it
2 **University of Wrocław, Poland**
piotr.wieczorek@cs.uni.wroc.pl

Abstract

In this chapter we illustrate the main techniques for processing queries in data integration. The first part of the chapter focuses on the problem of query answering in the relational setting, and describes approaches based on variants of the *chase*, along with how to deal with integrity constraints and access patterns. The second part of the chapter investigates query processing in the context of semistructured data, which is best described by graph-based data models, where the expressiveness of query languages not common in traditional database systems allows to point out the subtle differences between query answering and query rewriting. The chapter is closed by a very brief discussion of query processing in data integration with XML and ontologies.

1998 ACM Subject Classification H.2.4 [Systems]: Query processing, A.1 Introduction and survey

Keywords and phrases Query processing, data integration, relational data, semistructured data

Digital Object Identifier 10.4230/DFU.Vol5.10452.129

1 Introduction

The present chapter deals with the broad area of query processing in data integration, by illustrating the existing query answering techniques both for relational data and for semi-structured data.

The focus of the first part is on query answering in the relational case. We start with a brief description of possible results for various combinations of parameters of the problem. Then we describe the approaches that insist on the reconstruction of a representation of global database(s). The main algorithmic techniques are variants of the *chase* of the sources. We explain the concept of universal solution and show that it can be useful even if we would like to compute a query rewriting only (*inverse-rules method*). Next idea discussed is to rewrite the query such that it could be evaluated directly on the sources without materialisation (even if temporary) of the sources. We illustrate approaches based on the analysis of the relationship between the atoms of the user query and in the views, like in the MiniCon algorithm. We illustrate also the ways of dealing with integrity constraints and access patterns. We also discuss a technique of chasing the queries that leads to complete but unsound rewritings. The first part ends with a study of the information-theoretic notion of determinacy and its relation to rewriting.

Then, we investigate query processing in the context of semistructured data, capturing data that does not fit into the predefined and strict schemas of the relational setting, but is best described by graph-based data models. The main mechanism used for querying such kind of data consists in *regular path queries*, which are binary queries specifying the pairs of objects connected in a semistructured database by a path that conforms to a regular expression. Regular path queries can traverse the edges of a semistructured database in the forward direction only, but they can be extended with the ability of navigating edges also backward by means of an inverse operator, in which case we speak of *two-way regular path queries*. These query languages, not common in traditional database systems, possess a peculiar expressive richness that allows to uncover the subtle differences existing between query answering and query rewriting and which are completely blurred when focusing on conjunctive queries.

In the second part of the chapter, we first present a technique for rewriting regular expressions that can be readily applied for regular path queries, and we then discuss the relationship between query answering and query rewriting, also in relation to the relevant notion of *losslessness*. Indeed, we examine the relationship between various notions that assess different aspects of losslessness, w.r.t. answering and w.r.t. rewriting, in order to understand whether there is loss of information when processing a query based on a set of views and, in such a case, what is the cause.

We conclude the chapter by briefly discussing query processing in other data integration scenarios, namely XML data integration and the newly emerging area of ontology-based data integration. Due to space limitations, this part has no pretence of exhaustiveness, but it merely mentions current trends in data integration research.

2 Preliminaries

In this section, we introduce the necessary notation and give some basic definitions that will be used throughout the chapter.

An n-ary *relation* on a set A, where $n \in \mathbb{N}$ is called the *arity* of the relation, is a subset of the Cartesian product A^n, that is, a set of n-tuples of elements of A. A *signature* (or *alphabet*) is a finite set of relation symbols, each of which has an associated arity. A *relational structure* (or *instance*) over a signature σ is a pair $\mathbf{I} = \langle \Delta^{\mathbf{I}}, \cdot^{\mathbf{I}} \rangle$, where $\Delta^{\mathbf{I}}$ is a *domain* of objects and $\cdot^{\mathbf{I}}$ is a function associating each relation symbol $r \in \sigma$ with a relation $r^{\mathbf{I}}$ of appropriate arity (i.e., if r is an n-placed relation symbol, then $r^{\mathbf{I}}$ is an n-ary relation). A relational structure over a signature σ is also called a *σ-extension*. Sometimes we treat relational structures as sets of facts, that is, an instance $\mathbf{I}$ is a set containing exactly one fact $r(t)$ for each relation symbol r and each tuple $t \in r^{\mathbf{I}}$.

We call *database signature* a signature $\mathcal{R} = \{R_1, \ldots, R_n\}$ of *database symbols* and *view signature* a signature $\mathcal{V} = \{V_1, \ldots, V_k\}$ of *view symbols* not occurring in $\mathcal{R}$. Each view symbol $V \in \mathcal{V}$ has an associated *view definition* $V^{\mathcal{R}}$, which is a formula in some language $\mathcal{L}$ over $\mathcal{R}$ expressing V in terms of the database symbols. A $\mathcal{V}$-extension (i.e., a view instance) is denoted by $\mathbf{E}$; an $\mathcal{R}$-extension (i.e., a database instance) is denoted by $\mathbf{D}$ and simply called a *database*.

A *query* Q is a function from relational structures over a given signature $\mathcal{S}$ to relations, associating each relational structure $\mathbf{I}$ over $\mathcal{S}$ with a relation $Q(\mathbf{I})$ of a certain arity, called the *answer* to Q over $\mathbf{I}$. We refer to queries over the database signature $\mathcal{R}$ as *database queries* and to queries over the view signature $\mathcal{V}$ as *view queries*.

We consider two different assumptions on $\mathcal{V}$-extensions. Under Closed World Assumption (CWA), a $\mathcal{V}$-extension stores all the tuples that satisfy the view definitions. In this case we call the views *exact*. Alternatively, under Open World Assumption (OWA), a $\mathcal{V}$-extension may store only some of the tuples that satisfy the view definitions. In this case, we call the views be *sound*. We formally define sound and exact views as follows.

▶ **Definition 1.** Let $\mathbf{D}$ be a database and $\mathcal{V}^{\mathcal{R}}(\mathbf{D})$ be the $\mathcal{V}$-extension $\mathbf{E}$ such that $V(\mathbf{E}) = V^{\mathcal{R}}(\mathbf{D})$ for each $V \in \mathcal{V}$. A $\mathcal{V}$-extension $\mathbf{E}$ is said to be *sound* w.r.t. $\mathbf{D}$ iff $\mathbf{E} \subseteq \mathcal{V}^{\mathcal{R}}(\mathbf{D})$, and it is said to be *exact* w.r.t. $\mathbf{D}$ iff $\mathbf{E} = \mathcal{V}^{\mathcal{R}}(\mathbf{D})$.

In a traditional database setting, we answer queries by evaluating them on the database. In the context of view-based query processing, we have view extensions, and we aim at processing queries based only on the information about the views. There are two forms of view-based query processing, namely: view-based query answering (i.e., computing certain answers), and view-based query rewriting (i.e., computing query rewritings). The basic notions for the two tasks are defined as follows.

▶ **Definition 2** (Certain answers). The *certain answers* to a query Q *under sound views* $\mathcal{V}$ w.r.t. a $\mathcal{V}$-extension $\mathbf{E}$ is the set of all tuples t such that $t \in Q(\mathbf{D})$ for every database $\mathbf{D}$ w.r.t. which $\mathbf{E}$ is sound, that is:

$$\mathrm{cert}^{\mathrm{sound}}_{Q,\mathcal{V}}(\mathbf{E}) = \bigcap \bigl\{ Q(\mathbf{D}) \mid \mathbf{D} \text{ is such that } \mathbf{E} \subseteq \mathcal{V}^{\mathcal{R}}(\mathbf{D}) \bigr\} \ . \tag{1}$$

The *certain answers* to a query Q *under exact views* $\mathcal{V}$ w.r.t. a $\mathcal{V}$-extension $\mathbf{E}$ is the set of all tuples t such that $t \in Q(\mathbf{D})$ for every database $\mathbf{D}$ w.r.t. which E is exact, that is:

$$\mathrm{cert}^{\mathrm{exact}}_{Q,\mathcal{V}}(\mathbf{E}) = \bigcap \bigl\{ Q(\mathbf{D}) \mid \mathbf{D} \text{ is such that } \mathbf{E} = \mathcal{V}^{\mathcal{R}}(\mathbf{D}) \bigr\} \ . \tag{2}$$

▶ **Definition 3** (Rewriting). Let Q be a query over a database signature $\mathcal{R}$ and Q_{r} be a query over a view signature $\mathcal{V}$. Q_{r} is a *rewriting* of Q *under sound views* $\mathcal{V}$ iff for every database $\mathbf{D}$ and every $\mathcal{V}$-extension $\mathbf{E}$ which is sound w.r.t. $\mathbf{D}$ it holds that $Q_{\mathrm{r}}(\mathbf{E}) \subseteq Q(\mathbf{D})$. Q_{r} is a *rewriting* of Q *under exact views* $\mathcal{V}$ iff for every database $\mathbf{D}$ and every $\mathcal{V}$-extension $\mathbf{E}$ which is exact w.r.t. $\mathbf{D}$ it holds that $Q_{\mathrm{r}}\bigl(\mathcal{V}^{\mathcal{R}}(\mathbf{D})\bigr) \subseteq Q(\mathbf{D})$. A rewriting is *exact* if the subset inclusion in the above conditions is in fact an equality.

Rewritings are view queries that, in general, are formulated in a different language than the one used for database queries. We consider languages $\mathcal{L}_{\mathrm{q}}$ and $\mathcal{L}_{\mathrm{r}}$ in which database queries and rewritings are respectively expressed, along with a language $\mathcal{L}_{\mathrm{v}}$ for expressing view definitions. When the rewriting language $\mathcal{L}_{\mathrm{r}}$ is monotonic, the definition of rewriting under sound views coincides with the one of rewriting under exact views.

▶ **Proposition 1.** *Let $Q_r \in \mathcal{L}_r$ and $Q \in \mathcal{L}_q$, and let $\mathcal{L}_r$ be monotonic. Then, Q_r is a rewriting of Q under sound views $\mathcal{V}$ iff Q_r is a rewriting of Q under exact views $\mathcal{V}$.*

Proof. A rewriting under sound views is always also a rewriting under exact views, thus we only need to show the opposite direction (under the assumption that the rewriting language $\mathcal{L}_{\mathrm{r}}$ is monotonic). Assume that Q_{r} is a rewriting of Q under exact views $\mathcal{V}$, hence for every database $\mathbf{D}$ we have that $Q_{\mathrm{r}}\bigl(\mathcal{V}^{\mathcal{R}}(\mathbf{D})\bigr) \subseteq Q(\mathbf{D})$. By the monotonicity of $\mathcal{L}_{\mathrm{r}}$, we then get that for every $\mathcal{V}$-extension $\mathbf{E}$ such that $\mathbf{E} \subseteq \mathcal{V}^{\mathcal{R}}(\mathbf{D})$ it holds that $Q_{\mathrm{r}}(\mathbf{E}) \subseteq Q_{\mathrm{r}}\bigl(\mathcal{V}^{\mathcal{R}}(\mathbf{D})\bigr)$ and, in turn, $Q_{\mathrm{r}}(\mathbf{E}) \subseteq Q(\mathbf{D})$. ◀

Rewritings as defined above in Definition 3 are "contained" rewritings, that is, they provide an underestimation of the original query. Often we are interested in rewritings that are in a sense the best underestimations of the query, hence it is natural to consider rewritings that are maximal, also called *maximally-contained* rewritings, which we define below.

▶ **Definition 4** (Maximal rewriting). A rewriting $Q_{\mathrm{r}} \in \mathcal{L}_{\mathrm{r}}$ of a query $Q \in \mathcal{L}_{\mathrm{q}}$ under sound views $\mathcal{V}$ is *$\mathcal{L}_r$-maximal* iff, there is no rewriting $Q'_{\mathrm{r}} \in \mathcal{L}_{\mathrm{r}}$ of Q under $\mathcal{V}$ such that, for some $\mathcal{V}$-extension $\mathbf{E}$ which is sound w.r.t. some database $\mathbf{D}$, it is the case that $Q'_{\mathrm{r}}(\mathbf{E}) \supset Q_{\mathrm{r}}(\mathbf{E})$. A rewriting $Q_{\mathrm{r}} \in \mathcal{L}_{\mathrm{r}}$ of a query $Q \in \mathcal{L}_{\mathrm{q}}$ under exact views $\mathcal{V}$ is *$\mathcal{L}_r$-maximal* iff, there exists no rewriting $Q'_{\mathrm{r}} \in \mathcal{L}_{\mathrm{r}}$ of Q under $\mathcal{V}$ such that $Q'_{\mathrm{r}}\bigl(\mathcal{V}^{\mathcal{R}}(\mathbf{D})\bigr) \supset Q_{\mathrm{r}}\bigl(\mathcal{V}^{\mathcal{R}}(\mathbf{D})\bigr)$ for some database $\mathbf{D}$.

Dually, we can consider also *minimally-containing* rewritings that are the least overestimations of the query, but which are not even rewritings according to Definition 3. We study them in Section 3.

In some applications, it is of interest to consider rewritings that always provide exactly the same information provided by the original query. Such rewritings are called *exact* and are formally defined as follows:

▶ **Definition 5** (Exact rewriting). A rewriting $Q_\mathrm{r} \in \mathcal{L}_\mathrm{r}$ of a query $Q \in \mathcal{L}_\mathrm{q}$ under views $\mathcal{V}$ is *equivalent* to Q iff $Q_\mathrm{r}\big(\mathcal{V}^{\mathcal{R}}(\mathbf{D})\big) = Q(\mathbf{D})$ for every database $\mathbf{D}$.

That is, for every database $\mathbf{D}$, the evaluation of the rewriting on the view extensions w.r.t. $\mathbf{D}$ returns exactly the same answers given by evaluating the original query on $\mathbf{D}$. Observe that an exact rewriting may not always exist, possibly because it is not expressible in $\mathcal{L}_\mathrm{r}$.

We are now ready to provide the formal definition of data integration system, following the approach in [41]: A *data integration system* $\mathcal{I}$ is a triple $(\mathcal{G}, \mathcal{S}, \mathcal{M})$, where $\mathcal{G}$ is the *global schema*, $\mathcal{S}$ is the *source schema*, and $\mathcal{M}$ is the *mapping* between $\mathcal{G}$ and $\mathcal{S}$. The source schema describes the structure of the sources, where the real data is stored, while the global schema provides a reconciled, integrated and virtual view of the underlying sources. The source and global schema may be simple definitions of a set of relations or may allow for various forms of integrity constraints. The mapping connects the elements of the global schema with those of the source schema.

Consider a *source database* $\mathbf{D}$, that is, a relational structure over the source schema $\mathcal{S}$. We say that a *global database* $\mathbf{B}$ (any relational structure over $\mathcal{G}$) is legal for $\mathcal{I}$ and $\mathbf{D}$ if $\mathbf{B}$ satisfies all constraints of $\mathcal{G}$ and $\mathbf{B}$ satisfies the mapping $\mathcal{M}$ with respect to $\mathbf{D}$.

A mapping is expressed as a set of dependencies between $\mathcal{G}$ and $\mathcal{S}$. We consider dependencies in one of the following forms: *local-as-view* (LAV), *global-as-view* (GAV) and their combination *global-and-local-as-view* (GLAV).

In the GAV setting, the mapping characterise the content of each element of the global database as a view over the sources. Typically, GAV views are assumed to be exact, in which case the mapping (in the relational setting) can be specified as a set of dependencies of the form

$$\forall \vec{x}\ \varphi_{\mathcal{S}}(\vec{x}) \leftrightarrow R(\vec{x}) \ , \tag{3}$$

one for each relation symbol R in the global schema $\mathcal{G}$, where $\varphi_{\mathcal{S}}(\vec{x})$ is a query over $\mathcal{S}$. On the other hand, when GAV views are assumed to be sound, the equivalence in (3) must be replaced by an implication as follows:

$$\forall \vec{x}\ \varphi_{\mathcal{S}}(\vec{x}) \rightarrow R(\vec{x}) \ . \tag{4}$$

In the LAV setting, the sources are characterised in terms of a view over the global schema. Conversely, LAV views are commonly assumed to be sound, in which case the mapping can be specified as a set of dependencies of the form

$$\forall \vec{x}\ P(\vec{x}) \rightarrow \exists \vec{y}\ \psi_{\mathcal{G}}(\vec{x}, \vec{y}) \ , \tag{5}$$

one for each relation P in the source schema $\mathcal{S}$, where $\psi_{\mathcal{G}}(\vec{x}, \vec{y})$ is a query over $\mathcal{G}$. On the other hand, when LAV views are assumed to be exact, the implication in (5) must be replaced by an equivalence as follows:

$$\forall \vec{x}\ P(\vec{x}) \leftrightarrow \exists \vec{y}\ \psi_{\mathcal{G}}(\vec{x}, \vec{y}) \ . \tag{6}$$

Note that the LAV setting is exactly the scenario of view-based query processing. Indeed, in LAV we have one view for each source and the source content corresponds to the view extensions. A query posed against the global schema is answered by using both the view definitions (i.e., the mapping) and the view extension (i.e., the data at the sources). In some occasions, we refer to "view-based query processing" (i.e., query answering/rewriting using views), rather than to "query processing in LAV", so as to reflect the terminology used in the relevant literature.

Finally, we can consider the most general approach, called GLAV setting, which allows to specify mappings as any source-to-target tuple generating dependencies (s-t tgds) [34]:

$$\forall \vec{x}\ \varphi_{\mathcal{S}}(\vec{x}) \to \exists \vec{y}\ \psi_{\mathcal{G}}(\vec{x},\vec{y}) \ . \tag{7}$$

In general, given a data integration system and a source database, there could more than one legal global databases. In order to define what it means to answer a query in this case, we resort to the notion of certain answers. Similarly to what we did in the context of view-based query processing, we define the certain answers $\mathrm{cert}_{Q,\mathcal{I}}(\mathbf{D})$ to a query Q in the system $\mathcal{I}$ with respect to $\mathbf{D}$ as

$$\mathrm{cert}_{Q,\mathcal{I}}(\mathbf{D}) = \bigcap \big\{ Q(\mathbf{B}) \mid \mathbf{B} \text{ is a legal database for } \mathcal{I} \text{ and } \mathbf{D} \big\} \ . \tag{8}$$

In what follows, we will refer to the most well known query languages such as datalog, conjunctive queries (CQ) and their unions.

▶ **Definition 6.** *A datalog*$^\neg$ *rule* is an expression of the following form $P(\vec{x}) \leftarrow l_1(\vec{x}_1), \dots, l_n(\vec{x}_n)$, where each of $l_i(\vec{x}_i)$ is a *literal* i.e. positive or negated atom $R(\vec{x})$. We mention it explicitly if we allow negated atoms in queries by placing the superscript like in datalog$^\neg$. $P(\vec{x})$ is the head of the rule and $l_1(\vec{x}_1), \dots, l_n(\vec{x}_n)$ is the body. Each variable in the head of a rule must occur in the body of the rule. *A datalog*$^\neg$ *program* is a finite set of datalog$^\neg$ rules. If a query is given by one or more non-recursive rules and all of the rules have the same head then it is *a union of conjunctive queries (UCQ)*. A single, non-recursive, datalog rule is called *a conjunctive query (CQ)*. We say that a query is *safe* if, for each of its rules, every variable appearing in such rule (whether in the head or in the body) appears in a positive literal of the same rule.

When discussing conjunctive queries we will often use the notion of a canonical structure.

▶ **Definition 7** (Canonical structure). *A canonical structure* $\mathbf{C_Q}$ for a conjunctive query Q is the structure over the signature consisting of the relation symbols mentioned by Q such that each relation $R^{\mathbf{C_Q}}$ is the set of all tuples $\vec{a}$ of variables and constants in (positive) atoms $R(\vec{a})$ of Q.

▶ **Definition 8** (Expansion of a query). Let $\mathcal{R}$ be a database signature, let $\mathcal{V}$ be a view signature and assume that for each $V \in \mathcal{V}$ the view definition $V^{\mathcal{R}}$ is expressed as a conjunctive query. *An expansion* $\exp(Q_{\mathrm{r}})$ of a query Q_{r} over $\mathcal{V}$ is the query over $\mathcal{R}$ that is obtained by substituting every atom $V(\vec{a})$ in Q_{r} with the corresponding view definition $V^{\mathcal{R}}(\vec{a})$.

An expansion in a LAV data integration setting, where the mapping is given as a set of view definitions expressed as conjunctive queries, is defined analogously.

3 Query processing in relational data integration

In the first part of this section we will describe the main approaches that have been proposed for query answering in data integration for the case of relational data. Successively, we study the information-theoretic notion of determinacy and its relation to rewriting.

3.1 Approaches to query answering

There is a number of parameters that heavily influence query answering in data integration [41], namely: the presence of integrity constraints in the global schema, the class of allowed mappings, the classes of user queries, and the class of queries in the mappings.

Before we discuss the fundamental techniques in query answering, we review shortly the basic results for various combinations of the parameters.

GAV without constraints. This is the simplest case for query answering. We assume first-order queries in the mapping. If additionally, the views are exact then it can be proved that there is a single global database that is legal w.r.t the sources. This is the *retrieved global database* $\mathbf{B}$ over the source schema where the view extensions are computed using the view definitions. Note that since there are no existential variables on the right-hand side of view definitions, the tuples in $\mathbf{B}$ have the elements from the source database only.

Clearly, the answer to user query Q is computed by evaluating Q over the database $\mathbf{B}$.

Similarly, it is easy to modify the user query in order to obtain an equivalent query that can be executed over the sources. This can be done following the unfolding strategy where each of the atoms over V where V is a relation symbol in the global schema is substituted with the corresponding query in the GAV mapping (i.e. the view definition).

It turns out that if we assume that views are sound then only a very limited form of incompleteness appears. Namely, each view extension can be any superset of what is computed using the view definitions (for monotone queries). Thus, given a source database $\mathbf{D}$ there exist a number of global databases that are legal w.r.t. $\mathbf{D}$. However, it is easy to see that there exists the single minimal global database, which is the intersection of all such databases.

GAV with constraints. The presence of integrity constraints changes the situation radically. It turns out that in addition to incompleteness (i.e. there may be several global databases that are legal w.r.t. the sources) also the inconsistency can show up (i.e. there may be no global database that is legal w.r.t. sources).

Query answering in a GAV system with key and foreign key constraints has been studied in [11]. It is proved that computing certain answers corresponds to evaluating the query over a special *canonical database* that may be infinite in general. However, instead of direct evaluation of the query on the canonical database, the algorithm constructing a query rewriting in terms of a logic program is proposed. The approach results in polynomial data complexity for query answering in the case of conjunctive queries. The results were further extended in [13, 12] to deal with unions of conjunctive queries and inclusion dependencies.

(G)LAV. This is the setting that was the most intensively studied in literature, see [43, 1, 38, 33, 51, 48, 23]. Before we discuss the main approaches to (G)LAV query answering in detail we present a set of (data) complexity results in Tables 1 and 2 [1].

In the rest of the chapter we describe the main groups of query processing techniques for the case of LAV mappings. The approaches in the first group insist on the reconstruction of a representation of global database(s). The main algorithmic techniques are various variants of the *chase* of the sources. We explain the concept of universal solution and show that it can be useful even if we would like to compute a query rewriting only (*inverse-rules method*). Then, we describe approaches based on the analysis of the relationship between the atoms of the user query and in the views (e.g. the MiniCon algorithm). The main focus we put on the maximally-contained rewritings, since as proved in [1] for $\mathcal{L} \in \{\text{datalog}, \text{UCQ}\}$, $\mathcal{L}$-maximal

Table 1 Data complexity of computing certain answers assuming sound views [1].

views \ query	CQ	CQ$^{\neq}$	PQ	datalog	FO
CQ	PTIME	coNP	PTIME	PTIME	undec.
CQ$^{\neq}$	PTIME	coNP	PTIME	PTIME	undec.
PQ	coNP	coNP	coNP	coNP	undec.
datalog	coNP	undec.	coNP	undec.	undec.
FO	undec.	undec.	undec.	undec.	undec.

Table 2 Data complexity of computing certain answers assuming exact views [1].

views \ query	CQ	CQ$^{\neq}$	PQ	datalog	FO
CQ	coNP	coNP	coNP	coNP	undec.
CQ$^{\neq}$	coNP	coNP	coNP	coNP	undec.
PQ	coNP	coNP	coNP	coNP	undec.
datalog	undec.	undec.	undec.	undec.	undec.
FO	undec.	undec.	undec.	undec.	undec.

rewriting of a query in $\mathcal{L}$ under OWA computes exactly the set of certain answers. We illustrate also the ways of dealing with integrity constraints and access patterns. Finally, we discuss a technique of chasing the queries that leads to complete but unsound rewritings.

When describing complexity bounds we always mean *data complexity*, that is the complexity w.r.t. the size of data at the sources.

3.2 Reconstruction of global data

The first approach we are going to describe reduces query answering to the two steps:

1. Materialise some representation of global data.
2. Evaluate the query on the materialisation.

This approach is closely connected to the of *data exchange* [34] where we have a source schema, a target schema (that corresponds to the global schema in data integration) and a source-to-target mapping. The main difference between data integration and data exchange is in their goals. The primary goal of data exchange is to materialise the global database (*target instance*), while such materialisation is not required in data integration. Then query answering is performed using the materialised target instance without accessing the sources.

In this context we consider source-to-target dependencies of the form $\forall\vec{x}\forall\vec{y}\,\varphi(\vec{x},\vec{y}) \to \exists\vec{z}\,\psi(\vec{x},\vec{z})$, where φ is a conjunction of atoms over $\mathcal{S}$ and ψ is a conjunction of atoms over $\mathcal{G}$. Such dependencies correspond to the mappings in the GLAV data integration system.

The first step in order to implement this strategy is to find a good representation of possible global databases. Natural candidates has been studied in the field of *incomplete databases* such as Codd tables, naive tables or conditional tables [39, 37]. Generally, a table is a database such that in the domain, in addition to constants, variables (*nulls*) are allowed. A table represents a set of complete databases, each obtained by substituting all variables with constants.

Formally, let Δ be the set of all values, called *constants*, that may occur in domains of relational structures. In addition, the domain of a table can contain values from an infinite set $\underline{\mathrm{Var}}$ of labelled nulls. We have $\Delta \cap \underline{\mathrm{Var}} = \varnothing$. In naive tables there may be different occurrences of the same variable in contrast to a Codd tables where each variable can appear only once. While querying naive tables labelled nulls are treated like constants, in particular, each of them is equal to itself only. The distinction between variables and constants needs to be kept because of the function variables play when working with homomorphisms and substitutions.

A table $\mathbf{B}$ represents (under Open World Assumption) the following set of databases:

$$\text{Rep}(\mathbf{B}) = \{v(\mathbf{B}) \quad | \quad v \text{ maps all nulls in } \mathbf{B} \text{ to elements of } \Delta\},$$

where $v(\mathbf{B})$ is the database that results from $\mathbf{B}$ after replacing each null x with $v(x)$.

Given tables $\mathbf{B}$ and $\mathbf{B}'$, a homomorphism $h : \mathbf{B} \to \mathbf{B}'$ is a mapping from $\Delta^{\mathbf{B}}$ (i.e. the domain of $\mathbf{B}$), to $\Delta^{\mathbf{B}'}$ (i.e. the domain of $\mathbf{B}'$), such that (1) $h(c) = c$, for every $c \in \Delta$ and (2) for every fact $R(\vec{a})$ in $\mathbf{B}$, we have that $R(h(\vec{a}))$ is a fact of $\mathbf{B}'$.

The answer $Q(\mathbf{B})_{\downarrow}$ to a query Q over a table $\mathbf{B}$ is computed in the following way: first Q is evaluated on $\mathbf{B}$ (recall that in the process the nulls are treated as they were constants i.e. two nulls are equal if and only if they are syntactically equal) and then all tuples containing nulls are discarded.

Now we introduce the notion of *universal solution* [34]. Assume we are given a data integration system $\mathcal{I} = (\mathcal{G}, \mathcal{S}, \mathcal{M})$ and a source database $\mathbf{D}$. We say that a naive table $\mathbf{B}$ over the global schema $\mathcal{G}$ such that $\Delta^{\mathbf{B}} \subseteq \Delta \cup \underline{\text{Var}}$ is a universal solution for $\mathcal{I}$ and $\mathbf{D}$ if and only if (1) $\mathbf{B}$ is legal with respect to $\mathbf{D}$ and $\mathcal{I}$ and (2) for each global database $\mathbf{B}'$ that is legal w.r.t. $\mathbf{D}$ and $\mathcal{I}$ there exists a homomorphism $h : \mathbf{B} \to \mathbf{B}'$.

Why universal solutions are important? Intuitively, a universal solution is a solution that contains all the essential information as required by the mappings. Assume that the user queries are preserved under homomorphisms. That is, for every query Q if $h : \mathbf{B} \to \mathbf{B}'$ and $\vec{a} \in Q(\mathbf{B})$ then $h(\vec{a}) \in Q(\mathbf{B}')$. This condition is satisfied by positive queries such as conjunctive queries, unions of conjunctive queries and datalog.

We have the following theorem.

▶ **Theorem 9** ([34]). *Let $\mathcal{I}$ be a (GLAV) data integration system, let $\mathbf{D}$ be a source database, and let $\mathbf{B}$ be a universal solution for $\mathcal{I}$ and $\mathbf{D}$. For a query Q that is preserved under homomorphisms we have* $\text{cert}_{Q,\mathcal{I}}(\mathbf{D}) = Q(\mathbf{B})_{\downarrow}$.

Indeed, $\text{cert}_{Q,\mathcal{I}}(\mathbf{D}) \subseteq Q(\mathbf{B})_{\downarrow}$ because $\mathbf{B}$ is legal w.r.t. $\mathbf{D}$ and $\mathcal{I}$. Moreover $Q(\mathbf{B})_{\downarrow} \subseteq \text{cert}_{Q,\mathcal{I}}(\mathbf{D})$ because from the definition for each global database $\mathbf{B}'$ that is legal w.r.t. $\mathbf{D}$ and $\mathcal{I}$ there exists a homomorphism $h : \mathbf{B} \to \mathbf{B}'$ and h preserves Q.

In order to compute a universal solution we can use the classical chase [44, 4]:

▶ **Algorithm 1** (Chase the sources).

Input: *a data integration system $(\mathcal{G}, \mathcal{S}, \mathcal{M})$ and a source database $\mathbf{D}$.*

Output: *The canonical universal solution Sol($\mathbf{D}$).*

```
For every element of $\mathcal{M}$ of the form $\forall\vec{x}\forall\vec{y}\,\varphi(\vec{x},\vec{y}) \to \exists\vec{z}\,\psi(\vec{x},\vec{z})$
    for every tuple $(\vec{a},\vec{b})$ such that $\mathbf{D}$ satisfies $\varphi(\vec{a},\vec{b})$;
        insert the tuples $\psi(\vec{a},\vec{Z})$ into Sol($\mathbf{D}$), where $\vec{Z}$ is a fresh tuple of nulls
```

In the simple case where all dependencies are source-to-target and there are no constraints in $\mathcal{G}$ (i.e. no target dependencies) the chase always stops and constructs the solution in polynomial time.

The construction can be extended to the case where there are global constraints in $\mathcal{G}$, such as

- *equality-generating target dependencies (egds)* of the form

  $$\forall\vec{x}\,\varphi(\vec{x}) \to x_i = x_j,$$

  where φ is a conjunction of atoms over $\mathcal{G}$, x_i and x_j are among the variables in $\vec{x}$; and
- *tuple-generating target dependencies (tgds)* of the form

  $$\forall\vec{x}\forall\vec{y}\,\varphi(\vec{x},\vec{y}) \to \exists\vec{z}\,\psi(\vec{x},\vec{z}),$$

  where both φ and ψ are conjunctions of atoms over $\mathcal{G}$. If there are no existential variables in the right hand side then such tgds is called to be *full*.

During the chase, target egds and tgds are applied as long as $\mathrm{Sol}(\mathbf{D})$ does not satisfy all of them. However, in the presence of egds the chase may fail. This is because two distinct constants may be required to be equal. Moreover, in presence of tgds that are not full the chase does not always terminate and to make things worse checking termination of the chase is undecidable. [34] and [28] introduce a sufficient and verifiable in polynomial time condition for the termination of chase: the dependencies have to be *weakly-acyclic* (or have *stratified witnesses*). The idea is to keep track of how the values propagate during the chase, and, in particular, whether a new labelled null can determine the creation of another new labelled null, at a later chase step. For full details and proofs we refer to [34]. The research on the properties of the chase continues and better termination conditions are discovered (see e.g. [24].

Quite surprisingly, the concept of chase is useful even if we would like to compute a query rewriting only, without considering the view extensions in the first place. Consider the following strategy:

1. modify the query such that it is possible to evaluate it on the sources (i.e. compute the query rewriting),
2. evaluate the query rewriting on the sources.

Now we present *inverse-rules method* [33]. We assume the LAV setting $(\mathcal{G}, \mathcal{S}, \mathcal{M})$, in particular the mapping $\mathcal{M}$ is specified by the queries of the form $V(\vec{x}) \to \psi(\vec{x},\vec{z})$, where V is a symbol in $\mathcal{S}$ and $\psi(\vec{x},\vec{z})$ is a conjunction of atoms over $\mathcal{G}$. Furthermore, the language for user queries and query rewritings is datalog.

The *inverse rules* are defined as follows. For each rule r in $\mathcal{M}$ and for each of the free variables $y_1, \ldots, y_n$ in r we introduce a new function symbol f_{r,y_j} of the same arity as the head of r. Consider the rule r in $\mathcal{M}$ of the form

$$V(\vec{x}) \to \bigwedge_{i=1,\ldots,n} R_i(\vec{x_i}, \vec{z_i})$$

For $i = 1, \ldots, n$ we define the inverse rules in the following way

$$R_i(\vec{x_i}, \mathrm{s}(\vec{z_i})) \leftarrow V(\vec{x}).$$

The function s replaces each of the free variables z in $\vec{z_i}$ with the Skolem term $f_{r,z}(\vec{x})$.

▶ **Example 10.** The rule r

$$V(x,y) \to R_1(x,z) \land R_1(z,y)$$

generates the following inverse rules

$$R_1(x, f_{r,z}(x,y)) \leftarrow V(x,y),$$

and

$$R_1(f_{r,z}(x,y), z) \leftarrow V(x,y).$$

Intuitively, the Skolem terms play the role of nulls in the construction of a universal solution. Let $\mathcal{M}^{-1}$ be the set of inverse rules for the mapping $\mathcal{M}$. The rules of $\mathcal{M}^{-1}$ together with the rules from the original query Q form the desired query rewriting $(Q, \mathcal{M}^{-1})$ that can be evaluated over the sources. Note, however, that in the result there may be tuples containing function symbols. Clearly, in the final step all such tuples have to be discarded, similarly to the tuples with nulls in the process of naive evaluation above. We denote the resulting query as $(Q, \mathcal{M}^{-1})_{\downarrow}$.

Although the query $(Q, \mathcal{M}^{-1})_{\downarrow}$ is no longer a datalog query but rather a logic program (it contains function symbols) it can be evaluated in polynomial time w.r.t. size of a source database $\mathbf{D}$ [33]. Indeed, the evaluation has to be performed in two stages - it should start with the inverse rules (they introduce function symbols but they are not recursive) and then the original rules of Q should be applied (they could be recursive but they do not introduce function symbols).

Furthermore, it is possible to eliminate function symbols at all. This is because there are only finitely many function symbols in $(Q, \mathcal{M}^{-1})_{\downarrow}$ which makes it possible to encode them by introducing new predicate names in the datalog query. Therefore $(Q, \mathcal{M}^{-1})_{\downarrow}$ is expressible in datalog.

▶ **Theorem 11** ([33]). *Let $\mathcal{I} = (\mathcal{G}, \mathcal{S}, \mathcal{M})$ be a data integration setting, let $\mathbf{D}$ be a source database, and let $(Q, \mathcal{M}^{-1})_{\downarrow}$ be a query rewriting constructed as above. Then the rewriting $(Q, \mathcal{M}^{-1})_{\downarrow}$ is maximally-contained (i.e. it is a maximal rewriting in the class of datalog queries), it can be evaluated in polynomial time w.r.t. the size of the sources and it computes the certain answers, i.e.,* $\mathrm{cert}_{Q,\mathcal{I}}(\mathbf{D})$.

Note however, although $(Q, \mathcal{M}^{-1})_{\downarrow}$ is maximally-contained and it computes the certain answers it is not necessarily exact, that is, equivalent to the original query (see Definition 5). Actually, exactness is a joint property of both the rewriting and the data integration setting (see the discussion at the end of Section 4). Here, the problem of checking whether there exists an exact rewriting for a datalog query reduces to the datalog query containment [33] which is undecidable [53]. Nevertheless, $(Q, \mathcal{M}^{-1})_{\downarrow}$ is exact if an exact rewriting expressible either in datalog or as UCQ exists [1].

The inverse rules method can be extended to deal with (global) integrity constraints of the form of full dependencies. Recall that the full dependencies are dependencies of the form

$$\forall \vec{x} \varphi(\vec{x}) \rightarrow \psi(\vec{y}),$$

where necessarily $\vec{y} \subseteq \vec{x}$, φ is a conjunction of relational and equality atoms and ψ is a relation atom or equality atom. The key point is that there are no existentially quantified variables in the right hand side of a full dependency which guarantees the termination of chase. We construct a set chase(Γ) of datalog rules, one rule for each global full dependency in Γ. The idea is that the application of the rules in chase(Γ) will simulate the chase with (full) tgds and egds.

In presence of egds we need a way of enforcing equalities between variables and functional terms derived during the evaluation of the resulting query. We introduce a new relation E with an aim to capture equality. In order to be able to use E, the query and left hand sides of all dependencies have to be rewritten to their *rectified* version.

We limit ourselves to the following example.

▶ **Example 12.** Consider a query Q

$$\mathrm{Q}(x) \quad :- \quad P(c, x, y, y)$$

In order to use the equalities captured by E, Q is rewritten to its rectified version $\bar{\mathrm{Q}}$, where

- a constant c has to be replaced with fresh variable z and E has to include the pair (c, z),
- the variable x appears in the head, so it is replaced in the body with a fresh variable x' and E has to include (x, x'),
- the variable y appears more than once in Q, the second occurrence is replaced with a fresh variable y' and E has to include (y, y').

$$\bar{\mathrm{Q}}(x) \quad :- \quad P(z, x', y, y') \wedge E(x, x') \wedge E(c, z) \wedge E(y, y')$$

Consider a full dependency with the left hand side rectified:

$$P(x, y) \wedge P(y', z) \wedge E(y, y') \to P(x, z)$$

where P is a relation symbol in $\mathcal{G}$.

For each such dependency the following, new datalog rule is introduced in chase(Γ).

$$P(x, z) : -P(x, y) \wedge P(y', z) \wedge E(y, y')$$

Finally, the rewriting of a query Q is the union $Q \cup \mathcal{M}^{-1} \cup \mathrm{chase}(\Gamma) \cup \mathrm{equiv}(E)$, where equiv$(E)$ is the transitivity datalog rule for E: $E(x, y) \leftarrow E(x, z) \wedge E(z, y)$.

▶ **Theorem 13** ([33]). *Let $\mathcal{I} = (\mathcal{G}, \mathcal{S}, \mathcal{M})$ be a data integration setting, let $\mathbf{D}$ be a source database. We assume that the global schema $\mathcal{G}$ includes a set of full dependencies Γ. Consider a (rectified) query Q. Then $(Q, \mathcal{M}^{-1} \cup \mathrm{chase}(\Gamma) \cup \mathrm{equiv}(E))_{\downarrow}$ is a maximally-contained rewriting of Q.*

The restriction to the class of full dependencies guarantees evaluation of the query rewriting in polynomial time however it is more limiting than e.g. if we restrict dependencies to be weakly-acyclic. Note that we could introduce datalog rules for arbitrary tgds but then we have to deal with Skolem terms in recursive rules. Such logic programs need not terminate. Of course, this is nothing unexpected in the light of the results on the chase termination. Note, however, that there are examples of dependencies such that the chase terminates but the bottom-up evaluation of the logic program generated by them does not. E.g. consider the weakly-acyclic dependency $P(x_1, x_2) \to \exists y P(x_1, y)$ and the corresponding chase rule $P(x_1, f(x_1, x_2)) : -P(x_1, x_2)$.

We can also include special *domain enumeration* rules [33] to deal with access patterns on the views, however we defer it for a while.

3.3 Building the rewriting directly

The approaches to query answering that insist on the materialisation of global data, even if temporary as in the inverse rules method, share a serious disadvantage. The materialisation is not required in data integration and potentially it results in a lot of recomputation. In this section our goal is to modify the query in such a way that it is possible to evaluate it on the sources without materialising the global data.

First, recall that *an expansion* $\exp(Q_{\mathrm{r}})$ of a query Q_{r} over $\mathcal{V}$ is the query over $\mathcal{R}$ that is obtained by substituting every atom $V(\vec{a})$ in Q_{r} with the corresponding view definition $V^{\mathcal{R}}(\vec{a})$.

We present methods for computing maximally-contained query rewritings that are based on the analysis of how the atoms of a query can be mapped to the atoms of the expansion of the rewriting. Such mapping is called the *containment mapping* [21] and it necessarily exists since it witnesses the containment of the rewriting in the query.

We assume a LAV data integration system $(\mathcal{G}, \mathcal{S}, \mathcal{M})$ and that the user query Q and the queries in $\mathcal{M}$ are conjunctive queries (CQs). We say that an atom $R(\vec{x})$ of Q *is covered* by an atom $R(\vec{y})$ of a query $\mathcal{V} \in \mathcal{M}$ if there is a mapping $\theta : \vec{x} \to \vec{y}$ such that $\theta(\vec{x}) = \vec{y}$.[1] Intuitively, such partial mappings θ can be used to build the containment mapping.

We start with the following result

▶ **Theorem 14** ([42]). *Let $(\mathcal{G}, \mathcal{S}, \mathcal{M})$ be a LAV data integration system and let the user query Q and the queries in $\mathcal{M}$ be conjunctive queries (CQs). If the body of Q has n atoms, and Q' is a CQ-maximal rewriting (it contains all other rewritings that are CQ), then Q' has at most n atoms.*

The result follows from the fact that the atoms of the expansion $\exp(Q')$ must cover the atoms of Q. Since $\exp(Q') \subseteq Q$ then there is a homomorphism h from (the canonical structure for) Q to (the canonical structure for) $\exp(Q')$. Clearly, each atom in Q is mapped by h to at most one atom in $\exp(Q')$. Thus, if Q' contains more than n atoms then the expansion of at least one atom of Q' is disjoint from the image of h. Hence, the atom can be deleted and this contradicts the maximality of Q'.

This gives us the *brute-force* algorithm for the construction of maximally-contained rewriting in the class of unions of conjunctive queries. It is enough to consider all possible conjunctions of n or fewer atoms.

▶ **Theorem 15** ([42], [1]). *Let $(\mathcal{G}, \mathcal{S}, \mathcal{M})$ be a LAV data integration system, let Q and the queries in $\mathcal{M}$ be conjunctive queries (CQs) and let Q_r be the union of all CQ-maximal rewritings for Q. Then*
1. *Q_r is the UCQ-maximal rewriting,*
2. *Q_r can be evaluated in polynomial time w.r.t. the size of the sources,*
3. *Q_r is perfect (i.e. it computes exactly the certain answers),*
4. *Q_r is exact if an exact rewriting (in the class of UCQs) exists.*

The *bucket algorithm* [43] and its improved versions *MiniCon algorithm* [51], *SVB algorithm* [48] provide better ways of constructing maximally-contained rewritings. Let $(\mathcal{G}, \mathcal{S}, \mathcal{M})$ be a LAV data integration system and let Q and the queries (views) $\mathcal{V} = V_1, \ldots, V_n$ in $\mathcal{M}$ be conjunctive queries (CQs). The key idea is to analyse how the atoms of the expansion of a conjunctive rewriting can cover the atoms of the user query. We say that a variable is *distinguished* if it appears in the query head or in the head of a view definition. Existential variables are called *nondistinguished* (or *local*). We say that a variable in Q is *shared* if it it appears more than once in Q. There are a few simple conditions that a contained rewriting has to preserve.

C0 All atoms of Q have to be covered by atoms of the expansion of a rewriting.
C1 A distinguished variable in Q has to be mapped to a distinguished variable in some V_i in the expansion.
C2 A shared variable x of Q has to be mapped either to a distinguished variable or all atoms of Q involving x have to be covered by atoms in the expansion of a single V_i.

The example below illustrates the motivation for C2.

▶ **Example 16.** Consider a query $Q(x, z) \leftarrow P(x, y) \wedge P(y, z)$ and a mapping consisting of the two queries:

$$V_1(v) \to P(v, w), \qquad\qquad V_2(t) \to P(u, t)$$

[1] Such mapping does not exist only if some x appears twice in $\vec{x}$ (or because of constants if they are present).

Then the rewriting $V_1(x) \wedge V_2(z)$ is not contained in Q. This is because its expansion is $P(x,w) \wedge P(u,z)$, where w and u are fresh variables. Clearly, there is no way to equate the local variables w and u at the level of rewriting. Note that the variable y of Q is shared by two atoms in Q but since it is not in the head of Q it gets mapped to local variables of V_1 and V_2. Unfortunately, the bucket algorithm is not aware of the condition C2.

The bucket algorithm creates a bucket for each of the atoms of Q. The bucket for an atom $R(\vec{s})$ in Q contains the heads of the queries in $\mathcal{M}$ that include atoms to which $R(\vec{s})$ can be mapped. A query V_k can appear in the bucket for $R(\vec{s})$ multiple times if $R(\vec{s})$ can be mapped to more then one atom in V_k. Let $\vec{s} = s_1, \ldots, s_n$ and $\vec{t} = t_1, \ldots, t_n$.[2] An atom $R(\vec{t})$ in V_k is placed in the bucket for an atom $R(\vec{s})$ in Q if and only if:

- $R(\vec{s})$ and $R(\vec{t})$ unify (C0). [3]
- if x_i is in the head of Q then y_i has to be in the head of V_k (C1).

If both conditions are satisfied we insert into the bucket for $R(\vec{x})$ the new atom $\theta(\text{head}(V_k))$, where θ is the unifier and all variables of V_k not in the domain of θ (i.e. not mentioned in $R(\vec{y})$) are fresh.

After constructing the buckets the algorithm starts generating candidate rewritings. It considers every rewriting formed by a conjunction of atoms, one atom from each of the buckets. Effectively, it means that all elements of the Cartesian product of the buckets are enumerated. Note that because C0 and C2 are not enforced the algorithm has to check each of the rewritings whether either it is contained in Q or can be made to be contained by equating some of its variables. Finally, the resulting rewriting is the union of all contained conjunctive rewritings.

The bucket algorithm has two drawbacks - it generates a lot of candidate rewritings and then it performs an expensive containment test for each of them. This is because each of the atoms in Q is considered in isolation. On the other hand the MiniCon algorithm focuses on the interactions between the variables in the query and the mapping and enforces each of the conditions C0-C2. Again, let $(\mathcal{G}, \mathcal{S}, \mathcal{M})$ be a LAV data integration system and let Q and the views $V_1, \ldots, V_n$ in $\mathcal{M}$ be conjunctive queries (CQs). The algorithm constructs a kind of generalised buckets, called *MiniCon Descriptions (MCDs)*, however once it discovers that an atom of some V_i can cover an atom of Q it finds the minimal, additional set of query atoms that have to be covered together. Recall, that in the bucket algorithm we have a single bucket for each atom of Q and such buckets consist of atoms over V_i. Here, a MCD C is formed for a particular atom $V(h(\vec{x}))$, where h equates some variables of $\vec{x}$ (i.e. we may use $V(x,y,x)$ in the rewriting instead of $V(x,y,z)$). Formally, a MCD C for Q and V consists of

- a head homomorphism h,
- the atom $V(h(\vec{x}))$,
- a partial mapping θ from the variables of Q to the variables of V.
- a subset G_C of atoms of Q that are covered by some atom in the query $V(h(\vec{x}))$ using θ.

Note that the problem of bucket algorithm with enforcing the condition C0 is avoided by splitting each of the considered unifiers into a separate head homomorphism h and a mapping θ.

[2] Both $\vec{s}$ and $\vec{t}$ are tuples of constants and variables

[3] Actually, note that this does not guarantee that $R(\vec{t})$ can cover $R(\vec{s})$. The problem arises when the unifier equates a variable of $\vec{t}$ (in the head of V_k or not) with another variable of $\vec{t}$ that is not in the head of V_k (i.e. local variable). Then in the second phase if we use V_k in a rewriting we cannot enforce the equality since we have no access to the local variables at the level of rewriting. We verify this with the containment tests in the second phase.

The algorithm constructs a set $\mathcal{C}$ of MCDs in such a way that for each $C \in \mathcal{C}$, where $C = (h, V(h(\vec{x})), \theta, G_C)$ the conditions below hold

- for each head variable x in Q which is in the domain of θ, $\theta(x)$ is the head variable in $V(h(\vec{x}))$ (C1),
- If an existential variable x of Q is in the domain of θ and is shared by two or more atoms in Q then each of the atoms must be covered by $V(h(\vec{x}))$ (C2).
- The set of atoms G_C has to be the minimal one (i.e. it is not possible to cover a subset of the atoms in G by $V(h(\vec{x}))$ even if θ or h are extended).

The third condition, although not essential, allows for the optimisation of the second phase. Namely, the algorithm outputs a union of all rewritings that are combinations of MCDs $C_1, \ldots C_k$ such that the sets of covered atoms $G_{C_1}, \ldots, G_{C_k}$ form a partition of the set of atoms of Q. [4] Finally, it can be proved that the expansion of each of the rewritings is contained in Q. Hence, containment tests are not necessary.

3.4 Dealing with access patterns

Now, we show how to deal with a situation where access to data sources is limited. In practice, it may happen that some sources can answer only such queries where some variables are bound. For example, consider a source relation `owns(person, residence)` storing information about owners of residences. The source accepts queries where `residence` is bound to a given value (i.e. it is not a free variable). That is, users can ask who owns a given residence but is is forbidden to ask for a list of residences owned by a given person or for a list of all person-residence pairs.

In order to model such limitations we introduce *access patterns*. Formally, *an access pattern* for a k-ary relation V in the source schema is an expression V^α where α is a word of length k over the alphabet $\{\mathrm{i}, \mathrm{o}\}$, where 'i' stands for *input slot* (only bound values) and 'o' stands for *output slot* (no value required).

We say that a datalog query Q is *executable* if every variable of a rule appears first (when reading from left to right) in a positive atom in an output slot in the body of the rule. E.g. query $V^{iio}(0,0,x) \wedge V'^{i}(x)$ is executable, while query $V^{iio}(x,0,0)$ is not.

It turns out that, if we allow recursive rules in the rewriting, access patterns do not require special treatment in data integration system. Indeed, if recursion is allowed in the rewriting, then we can easily enumerate the whole active domain [33]. If recursion is not allowed, then we can simulate it with the chase, first transforming the domain enumeration rules into dependencies [23].

Given a source schema $\mathcal{S}$, let $\mathrm{domain}_\mathcal{S}$ be the unary recursive query with one rule of the form

$$\mathrm{domain}_\mathcal{S}(x_j) \leftarrow \mathrm{domain}_\mathcal{S}(x_{i_1}) \wedge \mathrm{domain}_\mathcal{S}(x_{i_2}) \wedge \ldots \wedge \mathrm{domain}_\mathcal{S}(x_{i_k}) \wedge V(\vec{x})$$

for each relation V in $\mathcal{S}$ with an access pattern α where $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ are in the input slots and x_j is in the output slot. Note that all-output access patterns give non-recursive rules and if all access patterns are input only then $\mathrm{domain}_\mathcal{S}$ is empty. Now a query $\mathrm{dext}(Q)$ for a query Q is given by the rules for $\mathrm{domain}_\mathcal{S}$ and the following rule

$$\mathrm{dext}(Q)(\vec{x}) \leftarrow \mathrm{domain}_\mathcal{S}(y_1), \ldots, \mathrm{domain}_\mathcal{S}(y_k) \wedge Q,$$

where the head of Q is $Q(\vec{x})$ and y_i are the variables in the body of Q.

Clearly, $\mathrm{dext}(Q)$ is executable and contained in Q.

[4] In particular the combinations cover mutually exclusive sets of atoms of Q.

Each rule of the query $\mathrm{domain}_{\mathcal{S}}$ as defined above can be captured with a constraint

$$\mathrm{domain}_{\mathcal{S}}(x_{i_1}) \wedge \mathrm{domain}_{\mathcal{S}}(x_{i_2}) \wedge \ldots \wedge \mathrm{domain}_{\mathcal{S}}(x_{i_k}) \wedge V(\vec{x}) \to \mathrm{domain}_{\mathcal{S}}(x_j).$$

For every relation V^α in $\mathcal{S}$ we define a constraint $\sigma_{V,\alpha}$

$$\mathrm{domain}_{\mathcal{S}}(\vec{x}) \wedge V(\vec{x}) \to V'(\vec{x}).$$

We denote all above constraints by $\mathcal{M}$. The constraints in $\mathcal{M}$ are over the source schema $\mathcal{S}$ and the global schema $\mathcal{G}$ but note that they are not source-to-target since $\mathrm{domain}_{\mathcal{S}}$ appears in both sides of some constraints.

▶ **Theorem 17** ([23]). *For a data integration setting $(\mathcal{G}, \mathcal{S}, \mathcal{M})$, where $\mathcal{S}$ is with access patterns, for any UCQ query Q and for a source database $\mathbf{D}$, the query $\mathrm{dext}(Q)(\vec{x})$ computes $\mathrm{cert}_{Q',(\mathcal{G},\mathcal{S},\mathcal{M})}(\mathbf{D})$ where Q' is obtained from Q by replacing every symbol V in $\mathcal{S}$ with the corresponding symbol V'.*

3.5 Chasing the query

Now we discuss the problem of rewriting queries using the views from a dual perspective. We follow ideas first presented in [25] (see also [26]). The approach results in query rewritings that are overestimations of the exact answers. Still, it turns out that such query rewritings can be computed with the use of the chase. This time, however, we chase the queries, and not the data as before.

Recall that in the most common setting LAV mappings consist of dependencies of the form

$$\forall \vec{x}\, V(\vec{x}) \to \exists \vec{y}\, \psi(\vec{x}, \vec{y}),$$

where V is a symbol in $\mathcal{S}$ and $\psi(\vec{x}, \vec{y})$ is a query over $\mathcal{G}$. Such dependencies enforce the view V to be sound. Here, we essentially make use of the implication in the other direction. Namely, the dependencies in the mapping are of the form

$$\forall \vec{x}, \vec{y}\, \psi(\vec{x}, \vec{y}) \to V(\vec{x})$$

where V is a symbol in $\mathcal{S}$ and $\psi(\vec{x}, \vec{y})$ is a query over $\mathcal{G}$. Such dependencies enforce the views V to be complete. Note that we may enforce exactness of views with the use of both kinds of dependencies at the same time. Recall also the result presented in Table 2 - computing certain answers under exact views is coNP-complete for CQ queries and views. Such a setting is called LAV with exact sources.

The following algorithm illustrates the idea of the approach limited to the case with conjunctive queries (CQs). Recall that by $\mathbf{C_Q}$ we denote the canonical structure for a CQ Q.

▶ **Algorithm 2** (ViewRewrite($(\mathcal{G}, \mathcal{S}, \mathcal{M}), Q(\vec{x})$)).

```
Input:
  - a LAV data integration setting $(\mathcal{G}, \mathcal{S}, \mathcal{M})$ with exact sources,
  - a conjunctive query $Q(\vec{x})$ over $\mathcal{G}$.
Let $Q'(\vec{x})$ be the empty query (with no atoms).
For each rule in $\mathcal{M}$ of the form $\forall \vec{x}, \vec{y}\, \psi(\vec{x}, \vec{y}) \to V(\vec{x})$
    for each tuple $\vec{a}$ such that
      - there is a tuple $\vec{b}$ such that $\mathbf{C_Q}$ satisfies $\psi(\vec{a}, \vec{b})$; and
      - $V(\vec{a})$ is not an atom of $Q'$.
     add $V(\vec{a})$ to the conjunction $Q'$ as a new atom.
$Q^{\mathcal{M}}(\vec{x}) := Q'$
Output: the query $Q^{\mathcal{M}}(\vec{x})$ over $\mathcal{S}$.
```

What can we say about the properties of $Q^{\mathcal{M}}$? Consider $\exp(Q^{\mathcal{M}})$, the expansion of $Q^{\mathcal{M}}$. Recall that $\exp(Q^{\mathcal{M}})$ is obtained by substituting every atom in $Q^{\mathcal{M}}$ with the corresponding view definition from $\mathcal{M}$. Clearly, in $\exp(Q^{\mathcal{M}})$ the symbols in $\mathcal{S}$ are replaced with the queries over $\mathcal{G}$. Now, note that there is always a containment mapping from $\exp(Q^{\mathcal{M}})$ to Q. Therefore, $Q^{\mathcal{M}}$ is complete but may be unsound. That is, it returns all tuples in $\mathrm{cert}_{Q,\mathcal{I}}(\mathbf{D})$ for every $\mathbf{D}$ but it may also return some incorrect tuples. Such rewritings are called *containing rewritings.* Furthermore, it can be shown that $Q^{\mathcal{M}}$ is contained in any other containing rewriting, thus it is the best containing rewriting possible. We call such rewritings *minimally-containing.*

We illustrate the algorithm with the following example.

▶ **Example 18.** Consider the query $Q(x_1, x_2) : -R(x_1, z_1) \wedge R(z_1, z_2) \wedge R(z_2, z_3) \wedge R(z_3, x_2)$ and the dependencies $R(y_1, a) \wedge R(a, y_2) \to V_1(y_1, y_2) \in \mathcal{M}_1$ and $R(y_1, z_1) \wedge R(z_1, z_2) \wedge R(z_2, y_2) \to V_2(y_1, y_2) \in \mathcal{M}_2$. Q is the path of length 4, V_1 is the path of length 2 and V_2 is the path of length 3. Then $Q^{\mathcal{M}_1}(x_1, x_2) : -V_1(x_1, z_2) \wedge V_1(z_1, z_3) \wedge V_1(z_2, x_2)$ and $Q^{\mathcal{M}_2}(x_1, x_2) : -V_2(x_1, z_3) \wedge V_2(z_1, x_2)$.

Note that $\exp(Q^{\mathcal{M}_1})$ is equivalent to Q while $\exp(Q^{\mathcal{M}_2})$ is not sound.

Finally, note that $\exp(Q^{\mathcal{M}})$ can be computed by the chase with the use of the rules with implications in the other direction (i.e. of the form $\forall \vec{x}\, V(\vec{x}) \to \exists \vec{y}\, \psi(\vec{x}, \vec{y})$.

The construction can be developed further, the paper [23] extends the result to UCQ$^{\neg}$ queries both in the mapping (i.e. view definitions) and as the user queries. Moreover, a broad class of integrity constraints in the global schema is allowed as well as the access patterns on views. The extension requires a careful technical development, but the basic idea of chasing the query remains and offers a unified treatment of the three flavours of rewriting problems: using views, with access patterns and under integrity constraints.

3.6 Determinacy and rewriting

The question of whether a query Q can be answered using a set of views $\mathcal{V}$ can be formulated at several levels: A language-specific formulation is given by the notion of query rewriting, while a more general formulation is given by the information-theoretic notion of *determinacy*, which is formally defined as follows.

▶ **Definition 19** (Determinacy). Given a set of views $\mathcal{V}$ and a query Q over a database signature $\mathcal{R}$, we say that $\mathcal{V}$ *determines* Q (written $\mathcal{V} \twoheadrightarrow Q$) iff, for every two database instances $\mathbf{D}_1$ and $\mathbf{D}_2$, we have that $Q(\mathbf{D}_1) = Q(\mathbf{D}_2)$ whenever $\mathcal{V}^{\mathcal{R}}(\mathbf{D}_1) = \mathcal{V}^{\mathcal{R}}(\mathbf{D}_2)$.

Intuitively, determinacy says that the views provide enough information to uniquely determine the answer to the query, but without specifying whether this can be done effectively or using a particular query language. Thus, the question of what is the relationship between determinacy and rewriting arises quite naturally. On the one hand, if a query Q has an exact rewriting Q_{r} under (exact) views $\mathcal{V}$, then $\mathcal{V} \twoheadrightarrow Q$; the converse, on the other hand, is in general not true, leading to the following definition.

▶ **Definition 20** (Complete rewriting language). A rewriting language $\mathcal{L}_{\mathrm{r}}$ is *complete* for $\mathcal{L}_{\mathrm{v}}$-to-$\mathcal{L}_{\mathrm{q}}$ iff $\mathcal{L}_{\mathrm{r}}$ can be used to express an exact rewriting of a query $Q \in \mathcal{L}_{\mathrm{q}}$ using views $\mathcal{V}$ defined in $\mathcal{L}_{\mathrm{v}}$ whenever $\mathcal{V} \twoheadrightarrow Q$.

An interesting case is when the language $\mathcal{L}_{\mathrm{q}}$ is itself complete for $\mathcal{L}_{\mathrm{v}}$-to-$\mathcal{L}_{\mathrm{q}}$, because in such a situation the query language needs not to be extended in order to take advantage of the available views. A thorough investigation of determinacy and its connection to rewriting is carried out in [49] for relational data, in a setting with exact views and exact rewritings where, for view languages $\mathcal{L}_\mathrm{v}$ and query languages $\mathcal{L}_\mathrm{q}$ ranging from first-order logic to conjunctive queries, the following questions are studied:

1. Is it decidable whether $\mathcal{V} \twoheadrightarrow Q$ for $\mathcal{V}$ in $\mathcal{L}_\mathrm{v}$ and Q in $\mathcal{L}_\mathrm{q}$?
2. Is $\mathcal{L}_\mathrm{q}$ complete for $\mathcal{L}_\mathrm{v}$-to-$\mathcal{L}_\mathrm{q}$ rewritings? If not, what is the minimal extension of $\mathcal{L}_\mathrm{q}$ in which such rewritings can be expressed?

Here, we summarise the main results for the following languages: First-order logic (FO); conjunctive queries (CQ) without equality, inequality and constants; unions of conjunctive queries (UCQ). For additional languages (e.g., existential FO) and further details see [49]. We start by reporting a general result establishing that determinacy becomes undecidable as soon as the query language $\mathcal{L}_\mathrm{q}$ is powerful enough so that satisfiability is undecidable, or as soon as the view language $\mathcal{L}_\mathrm{v}$ is such that validity is undecidable.

▶ **Proposition 2.** *If satisfiability of sentences in $\mathcal{L}_q$ is undecidable or validity of sentences in $\mathcal{L}_v$ is undecidable, then it is undecidable whether $\mathcal{V} \twoheadrightarrow Q$ for Q in $\mathcal{L}_q$ and $\mathcal{V}$ defined in $\mathcal{L}_v$.*

Clearly, a direct consequence of Proposition 2 is that determinacy is undecidable whenever queries or view definitions are expressed in FO. For what instead concerns rewritings, it turns out that in the unrestricted case, that is, when possibly infinite instances are allowed, FO is complete for FO-to-FO rewritings, but unfortunately this does not hold anymore when considering finite instances only. Indeed, in the restricted case, any language that is complete for FO-to-FO rewritings must express all computable queries.

Determinacy remains undecidable also for much weaker languages than FO. In fact, it has been shown in [49] that determinacy is undecidable for UCQs in a quite strong way, as the undecidability result holds for a fixed database schema and a fixed set of views. Concerning rewritings, since the question of whether a UCQ query has a UCQ rewriting in terms of a set of UCQ views is decidable [42], we also have that UCQ is not complete for UCQ-to-UCQ rewritings, otherwise we would get a contradiction of the undecidability of determinacy for UCQs mentioned above. Indeed, the following theorem from [49] shows that no monotonic language can be complete even for UCQ-to-CQ rewritings.

▶ **Theorem 21.** *Any language that is complete for UCQ-to-CQ rewritings must express non-monotonic queries.*

Proof. Consider the database signature $\mathcal{R} = \{r_1, r_2\}$ with r_1 and r_2 unary and the view signature $\mathcal{V} = \{v_1, v_2\}$ with the following view definitions:

$$v_1{}^{\mathcal{R}}\colon \exists u\, r_1(u) \wedge r_2(x) \;; \qquad\qquad v_2{}^{\mathcal{R}}\colon r_1(x) \vee r_2(x) \;.$$

Let Q be the query $r_2(x)$. It is easy to see that the views $\mathcal{V}$ determine the answer to Q. In fact, for any database $\mathbf{D}$, we have the following two cases:

- If $r_1(\mathbf{D}) \neq \varnothing$, then $\exists z\, r_1(z)$ is always true and $Q(\mathbf{D}) = v_1(\mathbf{D})$, that is, the answer to Q is provided by v_1;
- If $r_1(\mathbf{D}) = \varnothing$, then $v_2(\mathbf{D}) = r_1(\mathbf{D}) \cup r_2(\mathbf{D}) = r_2(\mathbf{D}) = Q(\mathbf{D})$, that is, the answer to Q is provided by v_2.

Therefore, $\mathcal{V} \twoheadrightarrow Q$. Now, let $\mathbf{D}_1$ be a database such that $r_1(\mathbf{D}_1) = \{a, b\}$ and $r_2(\mathbf{D}_1) = \varnothing$, and let $\mathbf{D}_2$ be a database for which $r_1(\mathbf{D}_2) = \{a\}$ and $r_2(\mathbf{D}_2) = \{b\}$. Then, we have that $v_1(\mathbf{D}_1) = \varnothing \subseteq \{a\} = v_1(\mathbf{D}_2)$ and $v_2(\mathbf{D}_1) = \{a, b\} = v_2(\mathbf{D}_2)$. However, $Q(\mathbf{D}_1) = \{a, b\} \not\subseteq \{a\} = Q(\mathbf{D}_2)$. Hence, the mapping that for each database $\mathbf{D}$ associates the query answer $Q(\mathbf{D})$ to the corresponding extension $\mathcal{V}^{\mathcal{R}}(\mathbf{D})$ is non-monotonic. ◀

The above proof shows that Theorem 21 holds even if the database relations, views and queries are restricted to be unary.

Whether a CQ query can be rewritten as another CQ query in terms of a set of CQ views is decidable [42]. Hence, if CQ were complete for CQ-to-CQ rewritings, we would immediately have a decision procedure also for the determinacy for CQ queries and views. However, in both [49] and [3] it is unfortunately shown that CQ is not complete for CQ-to-CQ rewritings and, indeed, the former also show that no monotonic language is. The proof in [3] is of particular interest in that it provides an infinite set of examples of CQ views and queries for which the views determine the query but the query has no CQ rewriting in terms of the views. The examples involve path queries $Q_n(x, y)$ on a binary relation r returning the pairs $\langle x, y\rangle$ for which there is an R-path of length n from x to y (for more information on path queries see Section 4). For instance, it can be shown that $\{Q_3, Q_4\} \twoheadrightarrow Q_5$ and that Q_5 has the following FO rewriting:

$$Q_5(x, y) \equiv \exists z\, Q_4(x, z) \wedge \forall v\, Q_3(v, z) \rightarrow Q_4(v, y) \;\; ,$$

but Q_5 has no CQ rewriting in terms of Q_3 and Q_4. For what concerns the decidability of determinacy for CQ views and queries, the problem remains open.

We conclude this section by presenting a well-behaved query language with respect to determinacy and rewriting, namely the so-called *packed fragment* (PF) of FO, identified and studied by Marteen Marx in [47]. We first make a short digression to introduce another fragment of FO, called the *guarded fragment* (GF), of which PF is a useful extension. GF is formally defined as the smallest set such that:

- it contains every first order atom over a given relational signature,
- it is closed under the boolean connectives, and
- it is closed under the following rule for quantified formulas: if $\phi(\overline{x}, \overline{y})$ is in GF, then so are also $\exists\overline{y}\,\big(G \wedge \phi(\overline{x}, \overline{y})\big)$ and $\forall\overline{y}\,\big(G \rightarrow \phi(\overline{x}, \overline{y})\big)$, provided that G is an atomic formula, called the *guard*, in which all variables $\overline{x}$ and $\overline{y}$ occur free.

In other words, GF allows only for quantified formulas of the form $\forall\overline{y}\,\big(G(\overline{x}, \overline{y}) \rightarrow \phi(\overline{x}, \overline{y})\big)$,[5] where G is an atomic relation symbol and $\phi(\overline{x}, \overline{y})$ is also in GF.

The key requirement in GF is that all of the variables occurring free in the subformula ϕ must also occur in the guard. On the one hand, such a restriction ensures nice logical properties to this fragment, in particular that the validity problem is decidable in 2EXPTIME; on the other hand, however, it strongly limits its expressivity. Intuitively, guarding corresponds to restricting the tuples that can be generated by queries to tuples whose elements form a sub-tuple[6] in some atomic relation, therefore views defined in GF always consist of sub-tuples of a relation in the database. For instance, we can define the view ICT-Employee(`name`) from the tables Employee(`emp`, `id`) and Department(`id`, `dep`) in the guarded fragment, as shown in the example below, but we cannot define the binary relation WorksFor(`emp`, `dep`) from these two tables.

▶ **Example 22.** The exact view definition

$$\forall x\, \big[\, \textsf{ICT-Employee}(x) \leftrightarrow \exists y\, \big(\textsf{Employee}(x, y) \wedge \textsf{Department}(y, \text{“ICT”})\big)\, \big]$$

[5] The variables in $\overline{x}$ are free, while the ones in $\overline{y}$ are universally quantified: when $\overline{x}$ is empty we have a GF closed formula; when $\overline{y}$ is empty we have an open formula without outermost quantification.

[6] That is, a projection over some of the elements in tuple.

is equivalent to (the conjunction of) the following two formulae:

$$\forall x, y \; \big[\big(\mathsf{Employee}(x, y) \wedge \mathsf{Department}(y, \text{“ICT”})\big) \to \mathsf{ICT\text{-}Employee}(x) \big] \;,$$
$$\forall x \quad \big[\mathsf{ICT\text{-}Employee}(x) \to \exists y \, \big(\mathsf{Employee}(x, y) \wedge \mathsf{Department}(y, \text{“ICT”})\big) \big] \;,$$

which are both in GF since the first one can be rewritten as

$$\forall y \; \big[\mathsf{Department}(y, \text{“ICT”}) \to \forall x \, \big(\mathsf{Employee}(x, y) \to \mathsf{ICT\text{-}Employee}(x)\big) \big] \;.$$

From a database perspective, GF coincides with the *semijoin algebra*, which is obtained from Codd’s relational algebra by replacing the product operator with the semijoin operator $\ltimes$ [40]. Indeed, differently from the natural join, the semijoin $R \ltimes S$ always returns a subset of the tuples in R. As the most common type of queries occurring in practise are CQs, it is interesting to remark that the guarded conjunctive queries (i.e., the CQs that are expressible in GF) are precisely the *acyclic conjunctive queries* [36]. A conjunctive query Q is acyclic iff so is its hypergraph, having as vertices all the variables occurring (free or bound) in Q and, for each atom $A(\overline{x})$ in Q, an hyperedge consisting of all the variables in $\overline{x}$. In turn, the hypergraph of a query is acyclic iff it can be reduced to the empty one by repeatedly deleting vertices that occur in exactly one hyperedge and deleting hyperedges that are contained in some other one [2].

The packed fragment extends GF by allowing guards of the form:

$$G(\overline{x}) = \bigwedge_k \exists \overline{y} \; A_k(\overline{x}, \overline{y}) \;, \tag{9}$$

where each A_k is an atom and, for every pair of distinct variables x_i and x_j, $G(\overline{x})$ contains an atom A_k in which x_i and x_j both occur free. A guard of this kind is a “safe product” whose hypergraph is such that any two of its vertices belong together to an hyperedge. When all the relation symbols are binary, the hypergraph is in fact a complete graph, that is, a clique. For example, the query $\mathsf{Employee}(x, z) \wedge \mathsf{Department}(z, y) \wedge \mathsf{WorksFor}(x, y)$ is a safe product and its hypergraph is indeed a clique, whereas $\exists z \, \big(\mathsf{Employee}(x, z) \wedge \mathsf{Department}(z, y)\big)$ is not because there is no atom connecting the variables x and y. The former query is expressible in PF but not in GF, as its hypergraph is cyclic. Indeed, PF is strictly more expressive than GF: the “until” operator of temporal logic is another example of a FO formula that can be expressed in PF but not in GF.

As for the positive properties, the validity problem in PF is 2EXPTIME-complete and every satisfiable PF formula is satisfiable on a finite model. The packed fragment enjoys good properties also w.r.t. determinacy and rewriting. In fact, as shown in [47], determinacy for PF queries and views is 2EXPTIME-complete and PF is complete for PF-to-PF rewritings. Moreover, the completeness result holds in addition for (unions of) packed conjunctive queries (PCQ), that is, formulas obtained from relation symbols and equality using only conjunction and existential quantification.[7] Indeed, we also have that PCQ is complete for PCQ-to-PCQ rewritings and UPCQ is complete for UPCQ-to-UPCQ rewritings. Other well-behaved classes of CQ queries and views, but orthogonal to PF, are reported in [49].

4 Query processing in semistructured data integration

Semistructured databases capture data that do not fit into rigid, predefined schemas, and are best described by means of graph-based data models. Indeed, a *semistructured database* is a

[7] Note that, although called packed conjunctive queries for simplicity, such formulas cannot always be written in prenex form due to syntactic restrictions.

Table 3 Summary of the major results on determinacy and completeness for rewritings.

Language $\mathcal{L}$	Determinacy	Complete for $\mathcal{L}$-to-$\mathcal{L}$ rewritings?	
		Finite	Unrestricted
FO	undecidable	✗	✓
UCQ	undecidable	✗	✗
CQ	open	✗	✗
PF	2EXPTIME-complete	✓	✓
(U)PCQ	2EXPTIME-complete	✓	✓

finite relational structure over a signature $\mathcal{R}$ of binary symbols, that can be seen as a finite directed graph where each node is an element of the domain and each edge is labelled by a symbol $r \in \mathcal{R}$. We denote by $r(x, y)$ an edge from x to y labelled by r, representing the fact that relation r holds between objects x and y. Clearly, in order to extract information from this kind of data model, special querying mechanisms are required that are not common in traditional database systems.

A *regular-path query* (RPQ) is a binary query defined in terms of a regular language over $\mathcal{R}$. In particular, the answer to an RPQ Q over a semistructured database $\mathbf{D}$ is the set $Q(\mathbf{D})$ of pairs of nodes connected in $\mathbf{D}$ by a directed path traversing a sequence of edges that form a word in the regular language $\mathcal{L}(Q)$ defined by Q. A *two-way regular-path query* (2RPQ) is an RPQ extended with the ability of traversing edges backwards when navigating in a semistructured database. Formally, a 2RPQ over $\mathcal{R}$ is defined in terms of a regular language over the alphabet $\mathcal{R}^{\pm} = \mathcal{R} \cup \{r^- \mid r \in \mathcal{R}\}$ obtained by adding, for each $r \in \mathcal{R}$, a new relation symbol r^- that denotes the *inverse* of r. In addition, the standard notion of path in a graph is replaced by the notion of "semipath", that is, a navigation of the database from some node to another according to a sequence of edge labels, in which edges are traversed forward or backward depending on whether the corresponding edge label is direct (i.e., some $r \in \mathcal{R}$) or inverse (i.e., some r^-). Then, the answer to a 2RPQ Q over a semistructured database $\mathbf{D}$ is given by the pairs of objects connected in $\mathbf{D}$ by a semipath conforming to the regular language defined by Q. Examples of RPQ and 2RPQ are given below.

▶ **Example 23.** Let $\mathbf{D}$ be the semistructured database over the signature $\mathcal{R} = \{r_1, r_2, r_3\}$ shown in Figure 1, and consider the RPQ $Q = (r_1 + r_2) \cdot r_3$ and the 2RPQ $Q' = (r_1 + r_2) \cdot r_3^-$ over $\mathcal{R}$. Then, the answers Q and Q' produce when evaluated over $\mathbf{D}$ are respectively given by $Q(\mathbf{D}) = \{\langle a, d\rangle, \langle c, d\rangle\}$ and $Q'(\mathbf{D}) = \{\langle a, b\rangle, \langle c, b\rangle\}$.

In this section, we study view-based query processing in the context of semistructured data. First, we focus on the query rewriting approach by presenting a method, proposed in

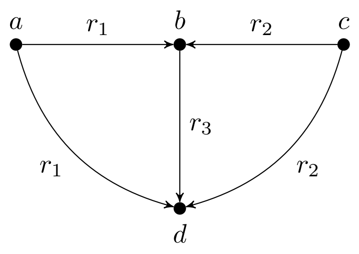


Figure 1 The semistructured database $\mathbf{D}$ of Example 23.

[17], for rewriting a query expressed as a regular expression (i.e., an RPQ) in terms of a set of views also expressed as regular expressions. Since RPQs are essentially regular expressions, the presented technique is directly applicable for them, and it is extended to 2RPQs in [20] using two-way automata to deal with inverse. Then, following [19], we explain and clarify the relationship between rewriting and answering, highlighting the distinction between them in the context of RPQs and 2RPQs. Indeed, the expressive richness of these languages allows to point out subtle differences between answering and rewriting that are blurred in the case of conjunctive queries. Lastly, we deal with the issue of trying to understand whether processing a query based on a set of views causes loss of information. We will introduce different notions for assessing "losslessness" w.r.t. answering and w.r.t. rewriting, and discuss the distinction and the relationship between them.

4.1 Rewriting of regular expressions

In this section, we present a method [17] for computing the rewriting of a regular expression (RE) in terms of other regular expressions. We consider a finite alphabet $\mathcal{R}$ of database symbols and a finite alphabet $\mathcal{V}$ of view symbols. Each view symbol $v \in \mathcal{V}$ is associated with a RE $v^{\mathcal{R}}$ over $\mathcal{R}$ that defines v in terms of the symbols in $\mathcal{R}$. Given a query Q expressed as a RE over $\mathcal{R}$, we want to reformulate it (if possible) by a suitable combination of the view symbols. The *expansion* of a language $\mathcal{L}$ over $\mathcal{V}$ is the language $\exp(\mathcal{L})$ over $\mathcal{R}$ consisting of all the words obtained from a word $v_1 \cdots v_n \in \mathcal{L}$ by substituting each v_i with every possible word of the regular language defined by the RE associated with v_i. In symbols:

$$\exp(\mathcal{L}) = \bigcup_{v_1 \cdots v_n \in \mathcal{L}} \{ w_1 \cdots w_n \mid w_i \in \mathcal{L}\big(v_i^{\mathcal{R}}\big) \} \ . \tag{10}$$

The expansion of a $\mathcal{V}$-word w is given by $\exp(\{w\})$.

Let Q_{r} be an expression defining a language $\mathcal{L}(Q_{\mathrm{r}})$ over $\mathcal{V}$: We say that Q_{r} is a *rewriting* w.r.t. $\mathcal{V}$ of a RE Q over $\mathcal{R}$ iff $\exp\big(\mathcal{L}(Q_{\mathrm{r}})\big) \subseteq \mathcal{L}(Q)$. A rewriting Q_{r} of Q w.r.t. $\mathcal{V}$ is said to be *exact* when $\exp\big(\mathcal{L}(Q_{\mathrm{r}})\big) = \mathcal{L}(Q)$. Obviously, we are interested in capturing the language defined by the original RE at best, that is, in finding rewritings that are maximal. Formally, a rewriting Q_{r} of Q w.r.t. $\mathcal{V}$ is $\mathcal{R}$*-maximal* iff every rewriting Q'_{r} of Q w.r.t. $\mathcal{V}$ is such that $\exp\big(\mathcal{L}(Q'_{\mathrm{r}})\big) \subseteq \exp\big(\mathcal{L}(Q_{\mathrm{r}})\big)$ and it is $\mathcal{V}$*-maximal* iff every rewriting Q'_{r} of Q w.r.t. $\mathcal{V}$ is such that $\mathcal{L}(Q'_{\mathrm{r}}) \subseteq \mathcal{L}(Q_{\mathrm{r}})$. Intuitively, for $\mathcal{V}$-maximality we compare the languages over $\mathcal{V}$ defined by the rewritings, while for $\mathcal{R}$-maximality we compare the corresponding expansions over $\mathcal{R}$ of such languages. Note that all of the $\mathcal{R}$-maximal (resp., $\mathcal{V}$-maximal) rewritings define the same language, and there exist $\mathcal{R}$-maximal rewritings which are not $\mathcal{V}$-maximal, as shown in the following example.

▶ **Example 24.** Let $\mathcal{R} = \{r\}$, $Q = r^*$ and $\mathcal{V} = \{v\}$ with $v^{\mathcal{R}} = r^*$. Then, we have that $Q_{\mathrm{r}} = v^*$ and $Q'_{\mathrm{r}} = v$ are both $\mathcal{R}$-maximal rewritings of Q w.r.t. $\mathcal{V}$, but the former is also $\mathcal{V}$-maximal while the latter is not.

As it turns out, $\mathcal{V}$-maximality is a sufficient condition for $\mathcal{R}$-maximality, i.e., every $\mathcal{V}$-maximal rewriting is also $\mathcal{R}$-maximal, hence we can search for a $\mathcal{V}$-maximal rewriting in order to find an $\mathcal{R}$-maximal one. This approach is suitable for (common) cases in which any $\mathcal{R}$-maximal rewriting would do, whereas it is not when one is interested in finding a specific $\mathcal{R}$-maximal rewriting (e.g., one maximising the use of some view symbol because it is less expensive), which might not indeed be $\mathcal{V}$-maximal.

We present a method that constructs a $\mathcal{V}$-maximal (hence also $\mathcal{R}$-maximal) rewriting of Q. Such a maximal rewriting always exists, even though it may be empty. The idea on which the proposed method is based consists in characterising, by means of an automaton, exactly those $\mathcal{V}$-words that do not belong to any of the languages defined by every possible rewriting of Q w.r.t. $\mathcal{V}$. Observe that a $\mathcal{V}$-word is such if its expansion contains an $\mathcal{R}$-word that is not in $\mathcal{L}(Q)$. Then, the complement of this automaton represents the maximal rewriting we are seeking, since it accepts exactly the $\mathcal{V}$-words whose expansions belong to $\mathcal{L}(Q)$.

▶ **Algorithm 3** (Compute maximal rewriting).

Input: *A regular expression Q over $\mathcal{R}$; a regular expression $v^{\mathcal{R}}$ over $\mathcal{R}$ for each $v \in \mathcal{V}$.*

```
1. Construct a deterministic automaton $A$ over $\mathcal{R}$ such that $\mathcal{L}(A) = \mathcal{L}(Q)$.
2. Define the automaton $A'$ over $\mathcal{V}$ such that:
   - $A'$ has the same set of states and the same initial state as $A$;
   - all states that are not final in $A$ are the final states of $A'$;
   - $A'$ has a transition from state $s_i$ to state $s_j$ labelled by $v$ iff there is a word in $\mathcal{L}(v^{\mathcal{R}})$
     that leads (through a sequence of transitions) from $s_i$ to $s_j$ in $A$.
3. Construct the complement $\overline{A'}$ of $A'$.[8]
```

Output: *The automaton $\overline{A'}$.*

The automaton A' built in the second step of the algorithm accepts only those $\mathcal{V}$-words leading from the initial state (which is the same as A's) to a state that is non-final for A (A's final states are non-final in A' and vice versa). The reason why the automaton A built in the first step is required to be deterministic is that we need to make sure that no $\mathcal{R}$-word in the expansion of each $\mathcal{V}$-word accepted by A' leads to a final state of A, thus belonging to $\mathcal{L}(Q)$.

▶ **Example 25.** Let $\mathcal{R} = \{r_1, r_2, r_3\}$ and $\mathcal{V} = \{v_1, v_2, v_3\}$, with the following definitions:

$$v_1{}^{\mathcal{R}} = r_1 \; ; \qquad v_2{}^{\mathcal{R}} = r_1 \cdot r_3{}^* \cdot r_2 \; ; \qquad v_3{}^{\mathcal{R}} = r_3 \; .$$

We want to rewrite the RE $Q = r_1 \cdot (r_2 \cdot r_1 + r_3)$ in terms of $\mathcal{V}$. The result of applying each step of Algorithm 3 is shown in Figure 2: first, we construct the deterministic automaton A of Figure 2a that accepts the language defined by Q; then, we build the automaton A' of Figure 2b that, for instance, has a v_2-labelled loop in s_0 because there exists a word (e.g., $r_1 \cdot r_2$) in the language defined by $v_2{}^{\mathcal{R}}$ leading from s_0 (through s_1) back to s_0 in A;[9] finally, since in this specific example A' is deterministic,[10] its complement $\overline{A'}$ is obtained by simply swapping final and non-final states of A' as in Figure 2c.

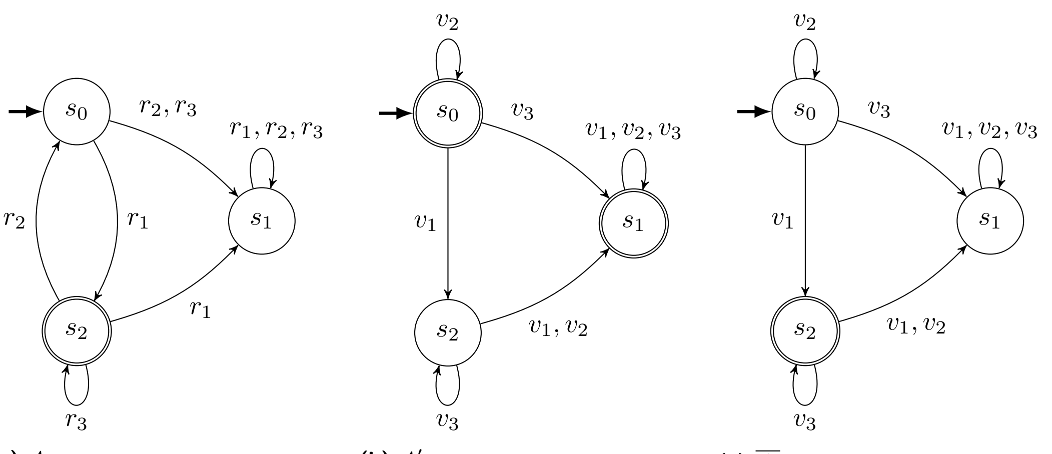
It can be proved (see [17] for details) that the automaton $\overline{A'}$ in the output of Algorithm 3 is indeed a $\mathcal{V}$-maximal rewriting of Q w.r.t. $\mathcal{V}$. This also shows that the language over $\mathcal{V}$ defined by the $\mathcal{V}$-maximal rewritings is in fact regular, although the form of the rewritings was not constrained a priori (the notion of rewriting was introduced as any expression defining a language over $\mathcal{V}$). The complexity of Algorithm 3 can be analysed as follows by considering the cost of each step:

1. Generating the deterministic automaton A from the regular expression Q is exponential.
2. Building A' is polynomial. In particular, it is required to check whether for each pair of states s_i and s_j there exists a word in the language defined by the RE associated with $v \in \mathcal{V}$ leading from s_i to s_j in A. This can be done by considering the automaton $A^{i,j}$,

[8] This is done by transforming A' into a deterministic automaton and swapping its final and non-final states. Obviously, the language $\mathcal{L}(\overline{A'})$ accepted by $\overline{A'}$ is the complement of the language $\mathcal{L}(A')$ accepted by A', that is, $\mathcal{L}(\overline{A'}) = \overline{\mathcal{L}(A')}$.

[9] In fact, in this particular example, **every** word in $\mathcal{L}(v_2{}^{\mathcal{R}})$ leads from s_0 back to s_0 in A.

[10] In general, the automaton A' constructed in the second step of Algorithm 3 is non-deterministic.



(a) A **(b)** A' **(c)** $\overline{A'}$

Figure 2 Construction of the rewriting of the regular expression $r_1 \cdot (r_2 \cdot r_1 + r_3)^*$ over $\mathcal{R} = \{r_1, r_2, r_3\}$ with respect to $\mathcal{V} = \{v_1, v_2, v_3\}$ where ${v_1}^{\mathcal{R}} = r_1$, ${v_2}^{\mathcal{R}} = r_1 \cdot {r_3}^* \cdot r_2$ and ${v_3}^{\mathcal{R}} = r_3$.

obtained from A by changing the initial state to s_i and the set of final states to $\{s_j\}$, and then checking for the non-emptiness of the product automaton between $A^{i,j}$ and an automaton for $\mathcal{L}(V^{\mathcal{R}})$.

3. Complementing A' (which is in general non-deterministic) is exponential.

Hence, generating the $\mathcal{V}$-maximal rewriting of a regular expression w.r.t. a set of regular expressions is in 2EXPTIME [17]. Deciding the existence of a non-empty rewriting can be done by first generating the maximal rewriting (2EXPTIME) and then checking for its non-emptiness (NLOGSPACE), which together give an EXPSPACE bound [17]. By means of a reduction from a suitable EXPSPACE-complete tiling problem, it has been shown that such a bound is tight (see [17] for the details of the reduction).

▶ **Theorem 26.** *The problem of verifying the existence of a non-empty rewriting of a regular expression w.r.t. a set of regular expressions is EXPSPACE-complete.*

Next, we present a method for checking whether a rewriting, in the form of the automaton $\overline{A'}$ obtained from Algorithm 3, is exact, that is, whether it captures the language defined by Q in its entirety.

▶ **Algorithm 4** (Verify exactness of rewriting).

Input: *The automatons A and $\overline{A'}$ built by Algorithm 3.*

1. *Construct an automaton B such that $\mathcal{L}(B) = \exp\big(\mathcal{L}(\overline{A'})\big)$ as follows:*
   - **a.** *For each $v_i \in \mathcal{V}$, construct an automaton A_i such that $\mathcal{L}(A_i) = \mathcal{L}({v_i}^{\mathcal{R}})$. We assume w.l.o.g. that each A_i has a unique initial state with no incoming edges and a unique final state with no outgoing edges.*
   - **b.** *B is obtained from $\overline{A'}$ by replacing each edge labelled by v_i with a new copy of A_i, identifying its initial state with the source of the edge and its final state with the target of the edge.*
2. *Check for the emptiness of $A \cap \overline{B}$.*

Output: *Yes, if $\mathcal{L}(A \cap \overline{B}) = \varnothing$; No, otherwise.*

It can be proved that the automaton $\overline{A'}$ is an exact rewriting of Q w.r.t. $\mathcal{V}$ iff $\mathcal{L}(A \cap \overline{B}) = \varnothing$. Thus, deciding the existence of an exact rewriting requires first the generation of $\overline{A'}$ (doubly

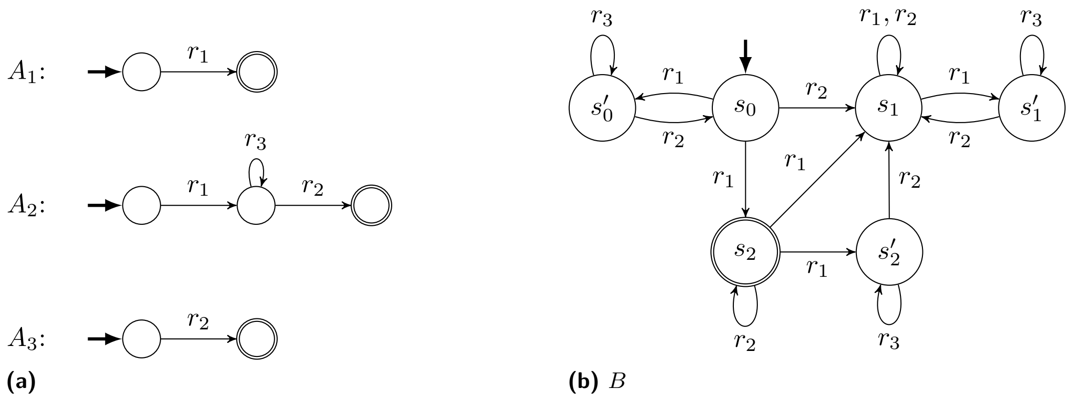


Figure 3 The construction of automaton B of Algorithm 4 in the setting of Example 25.

exponential), then the construction of B (polynomial) and its complementation (exponential), and finally checking whether $\mathcal{L}(A \cap \overline{B}) = \varnothing$ (NLOGSPACE). The explicit construction of $\overline{B}$ (and the further exponential blow-up it causes) can be fortunately avoided by building $\overline{B}$ "on-the-fly" as follows: Whenever the non-emptiness test for $A \cap \overline{B}$ requires to move from a state s_1 to a state s_2, the algorithm guesses s_2 and checks that it is directly connected to s_1; after guessing a suitable state, s_1 can be discarded; therefore, at each step, at most two states need to be kept in memory. By avoiding the generation of the whole $\overline{B}$, we obtain a 2EXPSPACE bound, while hardness can be shown with a reduction from a suitable tiling problem (see [17] for details), thus giving the following result.

▶ **Theorem 27.** *The problem of verifying the existence of an exact rewriting of a regular expression w.r.t. a set of regular expressions is 2EXPSPACE-complete.*

Referring back to Example 25, Figure 3 shows the construction of the automaton B, as in the first step of Algorithm 4. The automata A_1, A_2 and A_3 of Figure 3a accept the languages defined by the REs ${v_1}^{\mathcal{R}}$, ${v_2}^{\mathcal{R}}$ and ${v_3}^{\mathcal{R}}$, respectively. By substituting each v_i-labelled edge in $\overline{A'}$ (see Figure 2c) with a fresh copy of the corresponding A_i from Figure 3a, we obtain the automaton B depicted in Figure 3b. For instance, the loop labelled by v_2 in state s_0 of $\overline{A'}$ is replaced with a fresh copy of A_2, which requires the creation of a new state s'_0 in B and whose initial and final states are both identified with s_0. Similarly, replacing the v_2-labelled edge from s_2 to s_1 in $\overline{A'}$ requires the creation of a new state s'_2 in B, but this time the initial state of A_2 is identified with s_2 while the final one is identified with s_1. By applying the second step of Algorithm 4 it can be verified that the RE $Q_{\mathrm{r}} = {v_2}^* \cdot v_1 \cdot {v_3}^*$ represented by $\overline{A'}$ is indeed an exact rewriting of Q.

4.2 Answering, rewriting, and losslessness

Most of the work in the area of view-based query processing has focused on a setting based on conjunctive queries, in which it turns out that query answering and query rewriting coincide. Indeed, if the target queries are allowed to be written as UCQs, then the UCQ-maximal rewriting computes exactly the certain answers. For this reason, the relationship between answering and rewriting in view-based query processing is not always well understood. Calvanese et al. [19] provide a clean explanation of the distinction between the two notions in the context of semi-structured data, where the expressiveness of RPQs and 2RPQs allows to point out the subtle differences between answering and rewriting. Let us illustrate how

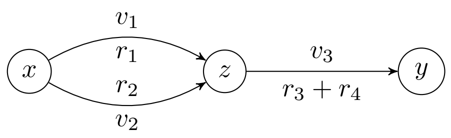


Figure 4 The structure of $\mathrm{cert}_{Q,\mathcal{V}}$ in Example 28.

things get more complicated when going beyond conjunctive queries by means of an example involving RPQs.

▶ **Example 28.** Let $\mathcal{R} = \{r_1, r_2, r_3, r_4\}$, $Q = r_1 \cdot r_3 + r_2 \cdot r_4$ and $\mathcal{V} = \{v_1, v_2, v_3\}$ with the following definitions:

$$v_1{}^{\mathcal{R}} = r_1 \ , \qquad v_2{}^{\mathcal{R}} = r_2 \ , \qquad v_3{}^{\mathcal{R}} = r_3 + r_4 \ .$$

By applying the techniques presented in the previous section, it can be checked that the RPQ-maximal rewriting of Q w.r.t. $\mathcal{V}$ is empty. On the other hand, the certain answers can be expressed as follows:

$$\mathrm{cert}_{Q,\mathcal{V}} = \big\{ \langle x, y \rangle \mid \exists z \; v_1(x,z) \wedge v_2(x,z) \wedge v_3(z,y) \big\} \ . \tag{11}$$

Observe that, as shown in Figure 4, (11) matches non-linear patterns in a database.

By characterising both answering and rewriting in terms of constraint satisfaction problems (see [19] for the technical details), Calvanese et al. show that the former is more precise than the latter. Indeed, while rewriting ignores that the same pair of objects might be connected by different edges, answering takes this into account, thus being able to recognise non-linear patterns as in the case of Example 28. Since it is defined directly in terms of the information content of the views, query answering is a more robust notion than query rewriting. In fact, whether a tuple is among the certain answers logically follows from the view extension. On the other hand, the motivation behind query rewriting is of a practical nature, that is, the need to access the view extensions by means of a specific query language.

A related and relevant issue in view-based query processing concerns *losslessness*, that is, the ability of being able to "completely" answer a query by relying only on the content of the views. Such an ability depends on the chosen approach to view-based query processing and can thus be considered from the query answering perspective, on the one hand, and from the query rewriting perspective, on the other. In the context of RPQs and 2RPQs, as we shall see, there is indeed a distinction between different notions of losslessness, while this is not the case for conjunctive queries, as answering and rewriting coincide.

A set of views $\mathcal{V}$ is *lossless* w.r.t. query Q iff for every database $\mathbf{D}$ we have that $Q(\mathbf{D}) = \mathrm{cert}_{Q,\mathcal{V}}\big(\mathcal{V}^{\mathcal{R}}(Q)\big)$. In other words, a set of views $\mathcal{V}$ is lossless when the certain answers over the $\mathcal{V}$-extension obtained from the view definitions always coincide with the answers to the query. This equivalence between the query and the certain answers determines whether the information content of a set of views is sufficient to answer completely a given query and constitutes the notion of losslessness w.r.t. answering.

In order to determine whether there is loss w.r.t. rewriting, we can compare rewritings to the original queries, on one side, and to the certain answers on the other, with the aim of checking for equivalence in either case. Depending on which comparison we are performing, we assess different aspects of losslessness w.r.t. query rewriting, which are represented by the two notions of *exactness* and *perfectness*. The former consists in the equivalence of a rewriting to the original query, modulo the view definitions; the latter is the equivalence to the certain answers. In other words, a rewriting is *exact* if it gives the same answers as the original query and *perfect* if it computes exactly the certain answers. In the case of conjunctive queries, where answering and rewriting coincide, maximal rewritings are always perfect and, as a consequence, also losslessness w.r.t. answering and losslessness w.r.t. rewriting coincide (that is, a set of views is lossless w.r.t. a query Q iff the maximal rewriting of Q is exact).

Let $\mathcal{V}$ be a set of 2RPQ views and Q a 2RPQ query, and denote by $Q_{\mathrm{r}}^{\max}$ the 2RPQ-maximal rewriting of Q with respect to $\mathcal{V}$. Then, using results from [18] and exploiting the fact that 2RPQs are monotone, we have that for every database $\mathbf{D}$ the following holds:

$$Q_{\mathrm{r}}^{\max}\big(\mathcal{V}^{\mathcal{R}}(\mathbf{D})\big) \subseteq \mathrm{cert}_{Q,\mathcal{V}}\big(\mathcal{V}^{\mathcal{R}}(\mathbf{D})\big) \subseteq Q(\mathbf{D}) \ . \tag{12}$$

Observe that in (12) we evaluate the certain answers and the maximal rewriting over the $\mathcal{V}$-extension $\mathcal{V}^{\mathcal{R}}(\mathbf{D})$ obtained from the view definitions, instead of an arbitrary $\mathcal{V}$-extension that is sound w.r.t. $\mathbf{D}$. The reason is that, when considering $\mathcal{V}$-extensions that are strict subsets of $\mathbf{D}$, there might clearly be loss of information, but such a loss would be unrelated to the "quality" of the views, which is what we are interested in assessing. The richness of 2RPQs allows us to discern and appreciate the differences between the various notions of losslessness, which correspond to the cases in which some or all of the inclusions in (12) are actually equalities. Each case is discussed below.

- When $\mathcal{V}$ is lossless w.r.t. Q, we have equivalence between the query and the certain answers, therefore the rightmost inclusion in (12) is an equality. In this case, there is no loss of information caused by the fact that we are answering the query based on a set of views (but there might still be loss due to rewriting).
- When $Q_{\mathrm{r}}^{\max}$ is perfect, that is, equivalent to the certain answers, we have that the leftmost inclusion in (12) is an equality. In this case, we do not lose answering power by resorting to rewriting (but there might still be loss due to the fact that we are answering a query based on a set of views).
- When $Q_{\mathrm{r}}^{\max}$ is exact, that is, equivalent to the query, we have that both inclusions in (12) are actually equalities. Therefore, the maximal rewriting is not only exact but also perfect and, in addition, the views are lossless w.r.t. the query. This means that exactness of the maximal rewriting is the strongest notion, in that it combines together both losslessness of the views and perfectness of the rewriting.

We conclude this section by briefly reporting the main complexity results obtained so far concerning losslessness, exactness and perfectness in the case of RPQs [17, 16] and 2RPQs [19, 20]. Let $\mathcal{V}$ be a set of RPQ views, Q an RPQ query and $Q_{\mathrm{r}}^{\max}$ the RPQ-maximal rewriting of Q with respect to $\mathcal{V}$. Then, the following hold:

- Checking whether $\mathcal{V}$ is lossless w.r.t. Q is EXPSPACE-complete in the size of Q and PSPACE-complete in the size of the view definitions $\mathcal{V}^{\mathcal{R}}$ [16].
- Verifying the existence of an exact rewriting of Q w.r.t. $\mathcal{V}$ is 2EXPSPACE-complete [17] (see Section 4.1).

Let $\mathcal{V}$ be a set of 2RPQ views, Q a 2RPQ query and $Q_{\mathrm{r}}^{\max}$ the 2RPQ-maximal rewriting of Q with respect to $\mathcal{V}$. Then, the following hold:

- Checking whether $\mathcal{V}$ is lossless w.r.t. Q can be done in EXPSPACE in the size of Q and the view definitions $\mathcal{V}^{\mathcal{R}}$ [19].
- Verifying the existence of an exact rewriting of Q w.r.t. $\mathcal{V}$ is 2EXPSPACE-complete [20].[11]
- Checking whether $Q_{\mathrm{r}}^{\max}$ is perfect can be done in N2EXPTIME in the size of Q and in NEXPTIME in the size of $\mathcal{V}^{\mathcal{R}}$ [19].

[11] In [20], the same complexity results reported in [17] for RPQs are indeed shown to hold also in the case of regular path queries with inverse.

5 Query processing in other data integration scenarios

We would like to conclude the chapter by briefly discussing query processing in the emerging areas of XML data integration and ontology-based data integration.

5.1 Data integration with XML

An increasing number of data integration applications uses XML for describing the global schema, which allows to hide proprietary source schemas that data owners do not want to disclose, while at the same time adhering to a newly-established standardised interface without the need of migrating existing data. Indeed, much of the work in data integration with XML is motivated by and focuses on the problem of publishing portions of relational and/or XML data stored in proprietary sources through a global XML schema against which user's queries are formulated. Compared to the case discussed in Section 4 of query processing in semistructured data, the translation of XML queries (expressed in XQuery) poses technical difficulties that do not arise with semistructured queries like RPQs and 2RPQs.

A methodology for integrating heterogeneous data sources under an XML global schema with LAV mapping is described in [45], where queries against the global schema, expressed in the XML query language XQuery, are translated into SQL queries over the local data sources. The approach followed in [45] allows for mixed relational and XML (in fact, any DOM-compliant) data sources and consists of the following three phases:

Normalisation The initial user's query (expressed in XQuery) is brought, through equivalence-preserving transformations, to a syntactical form that, whenever possible, can be directly translated to SQL. Indeed, normalisation identifies the features of XQuery that do not have SQL equivalents and filters out the queries which make use of them.

Translation The normalised query is translated into an SQL query over a generic, virtual, relational schema serving as an intermediate layer independent of the actual relationship between the XML global schema and the data sources. Indeed, the methodology described in [45] is implemented in the data integration system Agora [46], where relational and tree-structured data sources are defined by views over the global XML schema by means of such an intermediate schema that closely models the generic structure of an XML document.

Rewriting The translated SQL query on the generic schema is rewritten into a SQL query on the real data sources, by means of query rewriting algorithm [42] searching for maximally-contained rewritings. The authors of [45] argue that, even though in different scenarios a rewriting algorithm searching for exact query rewritings can be employed, as is the case in [46], in large-scale data integration applications, where there is no guarantee that all qualifying data is available, maximally-contained rather than exact rewritings are more appropriate.

The problem of finding an exact rewriting (if one exists) of an XQuery query against a global (public) XML schema into one or more queries over the source schema is studied in [28, 30] in a general setting with mixed (XML and relational) storage for the local (proprietary) data under the GLAV approach and in the presence of integrity constraints on both the global and the source schemas. The class of queries considered in [28] is a fragment of XQuery, consisting of so-called *XBind* queries whose general form is reminiscent of conjunctive queries; the constraints on the relational part are *disjunctive embedded dependencies* (DEDs) [2, 27] and on the XML part are *XML Integrity Constraints* (XIC) [31] whose expressive power captures a considerable part of XML Schema [9] including keys and "keyrefs" and more.

The approach followed in [28] consists in "compiling" an instance of the XML query rewriting problem into a relational one, and then solving the latter by using the Chase & Backchase (C&B) algorithm [25, 27]. The C&B enumerates the exact rewritings of a query that are "minimal" w.r.t. a set of constraints, in the sense that no atom can be removed from the rewriting without compromising equivalence to the original query (under the given constraints). Whenever the Chase is guaranteed to terminate, which is the case for sets of dependencies with *stratified witness* [28] (a.k.a. *weakly acyclic* sets of dependencies [34]), the C&B is complete, in the sense that outputs all (up to equivalence) and only the minimal rewritings of the input query under the given constraints.

The solutions devised in [28] are implemented in the MARS system [29], which can handle all of the cases handled by other LAV-based integration systems, such as Agora [46] and STORED [22] (for XML publishing), and Information Manifold [43] (for purely relational integration). Moreover, for what concerns XML publishing in the pure GAV approach, and when the storage schema is purely relational, MARS also subsumes the expressive power of the systems XPeranto [52] and SilkRoute [35].

Even though the settings of [28] and [45] differ in many aspects, as the former is GLAV-based, deals with exact rewritings and constraints are allowed on both global and source schemas, whereas the latter is LAV-based, deals with maximally-contained rewritings and no constraints are allowed, the two approaches are quite similar, in that they both make use of an intermediate relational schema, onto which the XML query rewriting problem is translated and then solved by means of relational query rewriting techniques.

We close our brief discussion of XML data integration by mentioning the work [55], where the authors address the problem of obtaining maximally-contained rewritings of XQuery queries in the presence of constraints on the XML global schema with LAV mapping, by devising a complete query rewriting algorithm that, differently from [45] and [28], operates directly on nested structures. The class of queries considered in [55] is a fragment of XQuery that includes nested subqueries; the constraints on the global schema are *nested equality-generating dependencies* (NEGDs), which include functional dependencies in relational or nested schemas, XML Schema key constraints, and more general constraints stating that certain tuples/elements must satisfy certain equalities.

Although [45] and [55] both use the LAV approach and deal with maximally-contained rewritings, the former does not allow for constraints whereas the latter takes into account constraints on the global schema. However, due to the translation to the intermediate generic relational schema, views and queries in [45] can be quite complex and hard to understand by humans, whereas the techniques in [55] operate directly at the XML level, thus resulting more natural and user-friendly.

5.2 Ontology-based data integration

The use of an ontology as the global schema in a data integration system, best known in the literature as ontology-based data access (OBDA), has the benefit of providing a semantically rich conceptual view of the information gathered by the system, which users can more easily understand. However, using an ontology to mediate the access to data sources amounts to data integration under integrity constraints, which must be fully considered during query answering, as they have a deep impact on how certain answers are computed [11]. Indeed, when the global schema is expressed in terms of even a very simple conceptual data model, the problem of incomplete information implicitly arises also in the GAV approach, making query processing (which without integrity constraints reduces to query unfolding) difficult [10].

When the global schema is an ontology expressed in the description logic ALCQI, which fully captures class-based representation formalisms, query answering in data integration is decidable [14]. However, the high computational data complexity makes the use of a such an expressive description logic infeasible in practice when dealing with large amounts of data, therefore the authors of [14] propose the adoption of DL-Lite, a specifically tailored restriction of ALCQI that ensures tractability of query answering in data integration while keeping enough expressive power. The "lightweight" description logic DL-Lite and its variants constitute a family of tractable description logics that are used in several applications, most notably the OBDA system MASTRO [15], providing access to heterogeneous relational data sources through an integrated ontology specified in a logic of the DL-Lite family.

Surveys on ontology-based approaches to semantic data integration present in the literature [54, 50, 7] compare several OBDA systems w.r.t. their reusability, changeability, and scalability. Recent systems, not considered in the above surveys, are OntoGrate [32] and the previously mentioned MASTRO [15], for relational data sources, and SOBA [8], for extracting and integrating information from heterogeneous resources including plain text, tables and image captions. We also cite MOMIS [6, 5] for the integration of semistructured data sources.

References

1 Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *Proceedings of PODS '98*, pages 254–263, 1998.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

3 Foto Afrati. Rewriting conjunctive queries determined by views. In Ludek Kucera and Antonín Kucera, editors, *Mathematical Foundations of Computer Science*, volume 4708 of *Lecture Notes in Computer Science*, pages 78–89. Springer Berlin / Heidelberg, 2007.

4 Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

5 Sonia Bergamaschi, Silvana Castano, and Maurizio Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, 28(1):54–59, March 1999.

6 Sonia Bergamaschi, Silvana Castano, Maurizio Vincini, and Domenico Beneventano. Semantic integration of heterogeneous information sources. *Data & Knowledge Engineering*, 36(3):215 – 249, 2001.

7 Agustina Buccella, Alejandra Cechich, and Rodríguez. *Encyclopedia of Database Technologies and Applications*, chapter Ontology-Based Data Integration, pages 450–456. Idea Group Reference, 2006.

8 Paul Buitelaar, Philipp Cimiano, Anette Frank, Matthias Hartung, and Stefania Racioppa. Ontology-based information extraction and integration from heterogeneous data sources. *International Journal of Human-Computer Studies*, 66(11):759–788, 2008.

9 Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Keys for XML. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 201–210. ACM, 2001.

10 Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Accessing data integration systems through conceptual schemas. In Hideko S.K̃unii, Sushil Jajodia, and Arne Sølvberg, editors, *Conceptual Modeling — ER 2001*, volume 2224 of *Lecture Notes in Computer Science*, pages 270–284. Springer Berlin / Heidelberg, 2001.

11 Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Data integration under integrity constraints. *Inf. Syst.*, 29(2):147–163, 2004.

12 Andrea Calì, Domenico Lembo, and Riccardo Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *PODS '03*, pages 260–271. ACM, 2003.

13 Andrea Calì, Domenico Lembo, and Riccardo Rosati. Query rewriting and answering under constraints in data integration systems. In *IJCAI*, pages 16–21, 2003.

14 Diego Calvanese and Giuseppe De Giacomo. Data integration: a logic-based perspective. *AI Magazine*, 26(1):59–70, March 2005.

15 Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. The MASTRO system for ontology-based data access. *Semantic Web*, 2(1):43–53, jan 2011.

16 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Lossless regular views. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 247–258, New York, NY, USA, 2002. ACM.

17 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Rewriting of regular expressions and regular path queries. *Journal of Computer and System Sciences*, 64(3):443–465, 2002.

18 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-based query containment. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '03, pages 56–67, New York, NY, USA, 2003. ACM.

19 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-based query processing: On the relationship between rewriting, answering and losslessness. *Theoretical Computer Science*, 371(3):169–182, 2007.

20 Diego Calvanese, Moshe Y. Vardi, Giuseppe de Giacomo, and Maurizio Lenzerini. View-based query processing for regular path queries with inverse. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '00, pages 58–66, New York, NY, USA, 2000. ACM.

21 Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

22 Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 431–442. ACM, 1999.

23 Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting queries using views with access patterns under integrity constraints. *Theoretical Computer Science*, 371(3):200–226, 2007.

24 Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In *PODS*, pages 149–158, 2008.

25 Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, pages 459–470, 1999.

26 Alin Deutsch, Lucian Popa, and Val Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

27 Alin Deutsch and Val Tannen. Optimization properties for classes of conjunctive regular path queries. In *Database Programming Languages*, pages 21–39. Springer, 2002.

28 Alin Deutsch and Val Tannen. Reformulation of XML queries and constraints. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory — ICDT 2003*, volume 2572 of *Lecture Notes in Computer Science*, pages 225–241. Springer Berlin / Heidelberg, 2002.

29 Alin Deutsch and Val Tannen. Mars: a system for publishing xml from mixed and redundant storage. In *Proceedings of the 29th international conference on Very large data bases*, volume 29 of *VLDB '03*, pages 201–212. VLDB Endowment, 2003.

30 Alin Deutsch and Val Tannen. XML queries and constraints, containment and reformulation. *Theoretical Computer Science*, 336(1):57–87, 2005.

31 Aline Deutsch and Val Tannen. Containment and integrity constraints for XPath. In *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB2001)*, volume 45 of *CEUR Workshop Proceedings*. ceur-ws.org, 2001.

32 Dejing Dou, Han Qin, and Paea Lependu. OntoGrate: Towards automatic integration for relational databases and the semantic web through an ontology-based framework. *International Journal of Semantic Computing*, 04(01):123–151, 2010.

33 Oliver M. Duschka, Michael R. Genesereth, and Alon Y. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, 43(1):49–73, 2000.

34 Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

35 Mary Fernández, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Trans. Database Syst.*, 27(4):438–493, December 2002.

36 Georg Gottlob, Nicola Leone, and Francesco Scarcello. On tractable queries and constraints. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications*, DEXA '99, pages 1–15. Springer-Verlag, 1999.

37 Gösta Grahne. Incomplete information. In *Encyclopedia of Database Systems*, pages 1405–1410. Springer US, 2009.

38 Gösta Grahne and Alberto O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *ICDT*, pages 332–347, 1999.

39 Tomasz Imieliński and Witold Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

40 Dirk Leinders, Maarten Marx, Jerzy Tyszkiewicz, and Jan Van den Bussche. The semijoin algebra and the guarded fragment. *Journal of Logic, Language and Information*, 14:331–343, 2005.

41 Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

42 Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. Answering queries using views. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '95, pages 95–104, New York, NY, USA, 1995. ACM.

43 Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, pages 251–262, 1996.

44 David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4(4):455–469, 1979.

45 Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML queries on heterogeneous data sources. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 241–250, 2001.

46 Ioana Manolescu, Daniela Florescu, Donald Kossmann, Florian Xhumari, and Dan Olteanu. Agora: Living with XML and relational. In *Proceedings of the International Conference on Very Large Data Bases*, pages 623–626, 2000.

47 Maarten Marx. Queries determined by views: Pack your views. In *Proceedings of PODS '07*, pages 23–30, 2007.

48 Prasenjit Mitra. An algorithm for answering queries efficiently using views. In *ADC*, pages 99–106, 2001.

49 Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *ACM Transactions on Database Systems*, 35(3):1–41, 2010.

50 Natalya F. Noy. Semantic integration: A survey of ontology-based approaches. *SIGMOD Record*, 33(4):65, 2004.

51 Rachel Pottinger and Alon Halevy. MiniCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2–3):182–198, 2001.

52 Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John Funderburk. Querying XML views of relational data. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 261–270. Morgan Kaufmann Publishers Inc., 2001.

53 Oded Shmueli. Equivalence of datalog queries is undecidable. *J. Log. Program.*, 15(3):231–241, 1993.

54 H. Wache, T. Voegele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-based integration of information – A survey of existing approaches. In *Proceedings of the IJCAI-01 Workshop on Ontologies and Information Sharing*, volume 47 of *CEUR Workshop Proceedings*, pages 108–117. ceur-ws.org, 2001.

55 Cong Yu and Lucian Popa. Constraint-based XML query rewriting for data integration. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 371–382. ACM, 2004.