Datalog and Its Extensions for Semantic Web Databases 4 h 16 18/7

Abstract. Since the early 70s, data management played a central role in organizations and represented a challenging area of research. A number of languages have been proposed to model, query, and manipulate data, as well as for expressing very general classes of integrity constraints, in-ference procedures, and ontological knowledge. Such languages are nowadays crucial for many applications such as semantic data publishing and integration, decision support, and knowledge management. In this tutorial we first introduce Datalog, a powerful rule-based language originally intended for expressing complex queries over relational data, and that today is at the basis of languages for the specification of optimization and constraint satisfaction problems as well as of ontological constraints in data and knowledge bases. We then discuss the limitations of Dat- alog for the semantic web, in particular for ontological modeling and reasoning, and we present several extensions that allow to capture some of the ontology languages of the OWL family, the standard language for semantic data modeling on the semantic web.

1 Introduction

Data and knowledge base systems played a dominant role in computer science since the 70s when organizations massively adopted them to support their business operations and decision making activities. Yet, in the last decade,

Datalog và các phần mở rộng của nó cho cơ sở dữ liệu web ngữ nghĩa

Tóm tắt. Kể từ đầu thập niên 70, quản lý dữ liệu đóng vai trò trung tâm trong các tổ chức và là một lĩnh vực nghiên cứu đầy thách thức. Các nhà nghiên cứu đã đưa ra một số ngôn ngữ để mô hình hóa, truy vấn và thao tác dữ liệu, cũng như biểu diễn các loại ràng buộc toàn vẹn rất tổng quát, các thủ tục suy diễn, và tri thức ontology. Hiện nay, những ngôn ngữ như thế đóng vai trò quan trọng trong nhiều ứng dụng chẳng hạn như xuất bản và tích hợp dữ liệu ngữ nghĩa , hỗ trợ quyết định, và quản trị tri thức. Trong bài hướng dẫn này, trước hết chúng tôi giới thiệu Datalog, một ngôn ngữ mạnh dựa trên quy tắc lúc đầu được thiết kế để biểu diễn các truy vấn phức tạp trên dữ liệu quan hệ, và hiện nay là cơ sở của các ngôn ngữ đặc tả tối ưu hóa và các bài toán thỏa mãn ràng buộc cũng như các ràng buộc ontology trong cơ sở dữ liệu và cơ sở tri thức. Sau đó, chúng ta sẽ đề cập đến những mặt hạn chế của datalog trong web ngữ nghĩa, đặc biệt trong mô hình hóa và suy luận ontology, và chúng tôi cũng trình bày một số phần mở rộng để chúng ta có thể hiểu được một số ngôn ngữ ontology thuộc họ OWL, ngôn ngữ tiêu chuẩn để mô hình hóa dữ liệu ngữ nghĩa trên web ngữ nghĩa.

such systems became even more popular as data and knowledge turn out to be an intrinsic part of every individual and collective activity in our society. In this setting, a major problem is to represent information in such a way that software programs can access it and act as if they really understand its semantics. On the one hand, initiatives such as the semantic web defined languages like RDF(S), FLORA and OWL to support the creation of semantically annotated data, enabling ontological querying and reasoning. On the other hand, the Linked Open Data (LOD) community produced very large amounts of semantically enriched data that enabled a multitude of data-driven semantic web applications. The need for efficient processing of semantic data stimulated several research initiatives addressing data management problems such as representation, storage and querying.

This tutorial is intended for people familiar with the basics of database and semantic web technologies who want to explore more in depth the connection between modeling languages used in these fields, and their practical adoption for knowledge representation and data management purposes.

A Bit of History. The data management problems we are facing today are not completely new. As an example, in the late 70s, Datalog [14] emerged as a prominent language from logic programming [30]. The term Datalog was coined by David Maier and reflects the intention of devising a counterpart of Prolog - the most prominent rule-

based formalism in computer science - for data processing. While Prolog is undecidable in general, if we consider the program as fixed, Datalog enjoys tractable reasoning complexity w.r.t. the size of the input database. Datalog's original aim was to be used as an expressive language for querying relational data; in fact, it adds recursion to the relational algebra, and therefore goes beyond the expressive power of select-project-join queries. Recursion is still important today for reasoning over complex paths in graphlike data which is abundant, for example, in the context of social networks and of the semantic web.

Applications of Datalog include data integration, reasoning about semistructured data, routing, security policy management, enterprise decision automation and many others. As a consequence, Datalog has evolved into a first-class formalism with efficient implementations such as DLV [21] and Clingo [25]. On the other hand, since Datalog rules (luật, quy tắc) are a representation of clauses in the function-free Horn fragment of first-order logic (FOL), Datalog revealed itself relevant also for semantic web applications such as ontological modeling and reasoning.

Example 1. Consider, as an example, the following Datalog rules expressing the knowledge that every female and every male is a person.

person(X) ^ female(X) person(X) ^ male(X).

Intuitively, to construct the set of all persons, we need to take into account the union of all females and males.

$$person(X) \leftarrow female(X) \qquad person(X) \leftarrow male(X).$$

In other words, some objects can be inferred to be persons, even without stating this fact explicitly. Datalog also provides a natural solution to some fundamental reasoning problems, such as the computation of the transitive closure of a binary relation, and is thus adequate for reasoning about graph reachability or connectedness. The rules
ancestor (X, Y) ^ parent (X, Y)
ancestor(X, Z) ^ parent(X, Y), ancestor(Y, Z)
express the ancestor relation between persons. Notice that in the last rule the predicate ancestor occurs in both sides of ^, which is an example of a recursive definition that is not possible in relational algebra. ■

From the other side, ontology languages for the semantic web were also designed to be able to express ontological information as the one above. In this setting, ontologies written in ontology languages are intended to describe and structure complex web resources, making them readily available for manipulation by automated agents [27,33]. The different ontology languages of the semantic web are based on the family of description logics (DLs) [5] which, in turn, are decidable fragments of FOL equipped with a convenient syntax.

DLs model a domain of interest in terms of concepts and roles, where concepts are interpreted as sets of individuals (i.e., constants), and roles as binary relations over them. A DL knowledge base K = (T, A) consists of a TBox T and an ABox A. The TBox T consists of axioms, where the most common axioms are

$$ancestor(X, Y) \leftarrow parent(X, Y)$$
$$ancestor(X, Z) \leftarrow parent(X, Y), ancestor(Y, Z)$$

statements of inclusion (a C 3) between pairs of concepts or roles. The ABox A is a set of facts about the participation of individuals in concepts and roles. In database terms, an ABox can be seen as a (possibly incomplete) relational database with unary and binary relations only, while a TBox is a set of expressive integrity constraints over the data.

As an example, consider the DL called SHIF, which has most of the features of the OWL languages. SHIF concepts are built by applying concept constructors on roles and other concepts. We use letters A, B,C for concept names and R, S for role names. A SHIF ABox A consists of ground atomic formulae of the form R(a, b) and A(a), where R is a role and A is a concept. A SHIF TBox T consists of axioms of one of the following forms:

(1) A n B C C is a concept inclusion. The complex concept A n B stands for conjunction of A and B, and the whole axiom states that each object that is both A and B is also C. This axiom translates into the FOL formula yX PA(X) A pB(X) ^ pC(X), where the predicates PA, pB, and pc represent the FOL predicates corresponding to the concept (resp., role) names. As an example, the axiom Parent n Male C Father can express the knowledge that male parents are fathers.

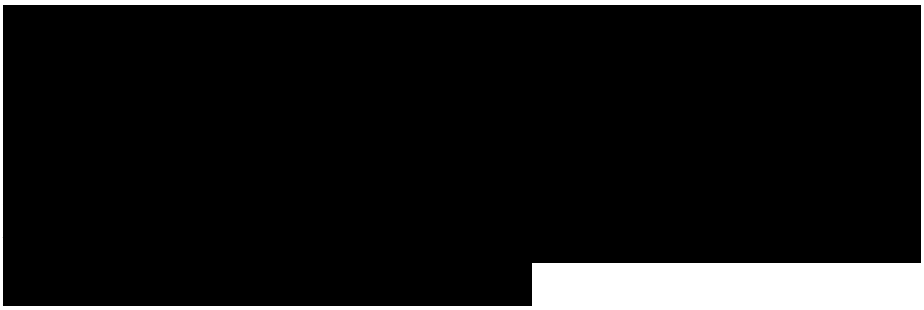
(2) A C B U C is another form of concept inclusion, stating that each object that is A must also be B or C. The axiom can be written as the FOL formula yX PA(X) ^ PB (X) Vpc

(X). As an example, the axiom Person C Female U Male states that every person must be female or male.

(3) A C yR.B is an inclusion employing on the right hand side a complex concept yR.B, denoting the set of all the objects such that all R "neighbors" are B, and is captured by yXyYpR(X,Y) ^ pB(Y). The whole axiom translates into the first-order logic formula yXVYPA(X) APR(X, Y) ^ pB (Y). An example of this axiom could be MetalDevice C yhasPart.Metal stating that all parts of a metal device are made of metal.

(4) A C 3R.B is an inclusion employing yet another kind of complex concept 3R.B on the right. 3R.B denotes the set of objects that have an R "neighbor" that is B, and is captured by yX BY pR (X, Y) A pB (Y). The full axiom translates into the formula VX PA(X) ^ BY pR(X,Y) A pB (Y). For example, we can use Student C Battends.Course to express the requirement that each student must attend at least one course.

(5) AC ^1 R.B restricts the number of R "neighbors" B can have, and can make parts of R functional (functionality constraints are common integrity constraints in databases). The axiom says that each object that is A can be related via the role R to at most one object that is B. This axiom translates into the formula VXVY1VY2 PA(X) APR(XY) Aps(Y1) APRXY) Aps(Y2) ^ Y1 = Y2. An example of this axiom could be PersonC ^1 hadldSocialSecNum, stating that a person can have at most one social security number.



(6) An axiom A disj B states disjointness between concepts A and B, e.g., Student disj Professor states that students and professors are disjoint sets. The axiom translates into the formula VX PA(X) A pB(X) ^ ±.

(7) R C S expresses the inclusion of R in S, e.g., brotherOf C relativeOf captures the knowledge that brothers are relatives. This axiom translates into the formula VXVY pR(X, Y) ^ pS(X, Y) of first-order logic.

(8) An axiom R inv S allows to define inverse roles, and is translated into first-order logic as VX VY pR(X, Y) ^ pS (Y, X). For example, a is child of b iff b is a parent of a. This can be expressed as parentOf inv childOf.

(9) Finally, the axiom trans(R) expresses transitivity of the role R, and is translated into the formula VXVYVZ pR(X, Y) ApR(Y., Z) ^ pR(X, Z).

Example 2. Consider the Datalog rules of Example 1, the same semantics can be (rather succinctly) expressed in DL syntax as

Female C Person Male C Person parentOf C ancestorOf trans(ancestorOf)

where (1,2) are concept inclusions, (3) is a role inclusion, and (4) states that the
role ancestor is transitive.

Datalog and DLs have several commonalities but also significant differences which need to be reconciled in order to make them interoperable in semantic web applications as also noticed in [35].

$$\mathsf{Female} \sqsubseteq \mathsf{Person} \quad (1)$$
$$\mathsf{Male} \sqsubseteq \mathsf{Person} \quad (2)$$
$$\mathsf{parentOf} \sqsubseteq \mathsf{ancestorOf} \quad (3)$$
$$\mathbf{trans}(\mathsf{ancestorOf}) \quad (4)$$

Differences include disjunction and existential quantification in DL ontologies, as well as the different assumptions underlying the semantics of the languages (e.g., open vs closed-world assumption). Table 1 shows a partial translation of SHIF into Datalog. As it can be seen, some expressions have no Datalog counterpart.

Table 1. From the DL SHIF to Datalog

(1) An axiom A C B U C expresses disjunctive information and thus cannot be directly expressed in plain Datalog. One possibility is to employ, bearing the computational cost, disjunctive Datalog, which does support rules of the form PB(X) VPC(X) ^ PA(X) [20].

(2) Notice that the translation of a functional constraint AC < 1 R.B into first-order logic involves the equality predicate. Due to the slight semantic differences between DLs and Datalog, equality is treated differently in the two settings. For instance, in contrast to Datalog, DLs do not employ the so-called unique name assumption (UNA). Different constants are treated as different domain objects by the Datalog semantics, yet a pair of constants may denote the same object in the standard semantics of first-order logic (and therefore DLs).

(3) A disjointness constraint A disj B can cause inconsistency of an ontology. E.g., the ontology K = (A, T), where A = {Blue(ba//), Red(ba//)} and T = {Red disj Blue}, is inconsistent, i.e., the first-order theory underlying K is unsatisfiable. Unfortunately, plain Datalog does

not have inconsistent programs and thus a satisfiability-preserving translation for disjointness constraints is not possible, in general.

(4) The axiom A C BR.B, whose translation into first-order logic involves existential quantification, exposes a crucial difference between Datalog and DLs. Datalog was designed and intended for reasoning over finite databases, under the assumption that only the objects explicitly mentioned in the database exist. In contrast, DL-based ontologies support existential quantification, and are thus able to refer to objects that are not explicitly named in the ontology.

Challenge. We note that existential quantification, disjointness and functional constraints play an important role in knowledge representation. They are in fact necessary to represent even simple constructs of common ontology languages, and there are many important DLs (such as the ones underlying the Lite and DL profiles of OWL) in which ontologies may have infinite and only infinite models that can not be captured by the models of a Datalog program.

Adding existential quantification to Datalog is the most interesting and relevant extension. Unfortunately, a naive extension of Datalog with existential quantified variables in the head is undecidable [3], and thus our task is nontrivial. We will try to identify meaningful fragments of Datalog with existential variables in the head that have sufficient expressiveness for the above applications, but still retain the

decidability of reasoning.

Due to the existential quantification, correct and terminating (hữu hạn, có kết thúc) reasoning algorithms require to develop methods for reasoning about infinite structures without explicitly building them. In the area of DLs and modal logic, decidability, algorithms and complexity results have been shown for various logics that do not enjoy finite models. Most of these fragments allow for only a limited number of variables (often two), and impose some form of guardedness, which can be roughly understood as a syntactically restricted form of quantification that only allows to talk about relations between objects that are close to each other in a structure, and results in regular models that are conveniently similar to trees. While guardedness is of course a limitation, it is often claimed to be a robust reason for decidability [26,39]. Furthermore, there is wide evidence suggesting that guardedness is not overly restrictive for many knowledge representation problems, and it is implicitly or explicitly present in many of the popular languages. Inspired by the above considerations, we will present guarded extensions of Datalog as a solution to our problems.

Roadmap. In the next section we provide some preliminary notions about relational databases and queries. Syntax and semantics of Datalog is presented in Section 3, while Section 4 shows how Datalog can be provided with additional features to capture some of the most common DLs of the semantic web.

In particular, we overview Datalog± [9], a family of extensions to Datalog that can be used to specify ontological constraints over databases while preserving tractability of reasoning w.r.t. the size of the data. Finally, Section 5 draws some concluding remarks and collects several useful pointers to further reading material.

2 Preliminaries

In this section, we briefly recall some basics on relational databases, and (Boolean) conjunctive queries.

We define the following pairwise disjoint (infinite) sets of symbols: a set r of constants, that constitute the "normal" domain of a database, a set rN of labeled nulls (giá trị không, rỗng), used as placeholders for unknown values, and thus can be also seen as (globally) existentially-quantified variables, and a set rV of (regular) (chính quy) variables, used in queries and dependencies. Different constants represent different values (unique name assumption), while different nulls may represent the same value. A lexicographic order is defined on r U rN, such that every value in rN follows all those in r. We denote by X sequences (or sets, with a slight abuse of notation) of variables or constants Xi,..., Xk, with k ^ 0. Throughout, let [n] = {1,..., n}, for any integer n ^ 1.

A relational schema R (or simply schema) is a set of relational symbols (or predicates), each with its associated arity. A position r[i], in a schema R, is identified by a predicate r G R and its i-th argument. A term t is a constant, null, or variable. An

atomic formula (or simply atom (nguyên tố, nguyên tử)) has the form r(t1,..., tn), where r is an n-ngôi predicate, and 11,..., tn are terms. For an atom a, we denote dom(a) and var(a) the set of its constants and variables, respectively; these notations are naturally extended to sets of atoms. Conjunctions of atoms are often identified with the sets of their atoms. A relational instance (or simply instance) I for a schema R is a (possibly infinite) set of atoms of the form r(t), where r is an n-ary predicate of R, and t G (r U rN)n. We denote as r(I) the set {t | r(t) G I}. A database D for a schema R is a finite relational instance for R such that dom(D) C r.

A conjunctive query (CQ) q of arity n over a schema R, written as q/n, is an assertion the form p(X) ^ ¥>(X, Y), where y>(X, Y), called the body of q and denoted body(q), is a conjunction of atoms over R having as arguments constants of r and variables of rV, and p is an n-ary predicate not occurring in R. A Boolean conjunctive query (BCQ) is a CQ of arity zero. Answers to queries are defined via homomorphisms. A homomorphism h from a set of atoms A1 to a set of atoms A2 is a mapping h : r U rN U rV ^ r U rN U rV such that: t G r implies h(t) = t, and r(t1,... ,tn) G A1 implies h(r(ti,... ,tn)) = r(h(t1),..., h(tn)) G A2. The answer to a CQ q/n over an instance I, denoted q(I), is the set of all n-tuples t G rn for which there exists a homomorphism h such that h(y>(X, Y)) C I and h(X) = t. A BCQ has only the empty tuple {) as possible answer, in which case

we say it has a positive answer. Formally, a BCQ has positive answer over I, denoted I = q, iff {) G q(I), or, equivalently,
q(I) = 0.

3 Datalog

Datalog (see, e.g., [1,14]) has been used as a paradigmatic database programming and query language for over three decades. While it is rarely used directly as a query language in corporate application contexts, Datalog has influenced the development of popular query languages such as SQL, whose newer versions allow one to express recursive queries.

3.1 Syntax

A Datalog rule p is an expression of the form where n ^ 0, a0,... ,an are atoms over a relational schema which contain constants of r and variables of IV, and each variable occurring in a0 must appear in at least one of a1;... ,an. The atom a0 is called the head, denoted head(p), while the set of atoms {a1;..., an} is called the body, denoted body(p). In other terms, a Datalog rule is a function-free Horn clause. A Datalog program P is a finite set of Datalog rules. The set of constants occurring in P is denoted dom(P). An extensional predicate is a predicate that does not occur in the head of any rule of P, while an intensional predicate is a predicate that occurs in the head of some rule of P. The extensional (database) schema of P, denoted EDB(P), consists of all the extensional predicates of P, whose values are given via an input database, while the intensional schema of P, denoted IDB(P),

consists of all the intensional predicates of P, whose values are computed by the program. The schema of P, written SCH(P), is the set of predicates EDB(P) U IDB(P). As we shall see, the semantics of a Datalog program is a mapping from databases for EDB(P) to databases for IDB(P).

Example 3. Consider the Datalog program Pgraph constituted by

sp.reachable(X) sp-node(X)

sp-reachable (Y) edge{X,Y), sp-'reachable(X),

which takes as input EDB a directed graph given by a binary edge relation (hệ thức), plus a set of special nodes of this graph given by a unary relation sp-node. Clearly,

EDB{Pgravh) = {edge, sp-node} and IDB(Pgraph) = {sp-reachable}.

The above (recursive) program computes the set sp-reachable of all vertices in the graph which are reachable from special nodes. ■

3.2 Semantics

An interesting and elegant property of Datalog is the fact the there are three different but equivalent approaches to defining its semantics: a model-theoretic approach where the Datalog rules are considered as logical sentences asserting a property of the desired result, a fixpoint (điểm cố định, dấu chấm cố định) approach where the semantics are defined as a particular solution of a fixpoint equation, and a proof-theoretic approach which is based on obtaining proofs of facts. In the sequel, we discuss the model-theoretic and fixpoint semantics; for details on the proof- theoretic approach we refer the interested

$$sp\_reachable(X) \leftarrow sp\_node(X)$$
$$sp\_reachable(Y) \leftarrow edge(X,Y), sp\_reachable(X),$$

$$EDB(P_{graph}) = \{edge, sp\_node\} \text{ and } IDB(P_{graph}) = \{sp\_reachable\}.$$

reader to [1, Chapter 12, Section 12.4].

Model-Theoretic Semantics. The idea underlying this approach is to consider the given Datalog program as a set of first-order sentences (also called a first-order theory) which describes the desired outcome of the program. More precisely, to a rule p '. a0 <— a1;..., an we associate the first-order sentence

VXi... VXyji (aq ^A ... A

[redacted]

$$\forall X_1 \ldots \forall X_m \, (\underline{a}_0 \leftarrow \underline{a}_1 \wedge \ldots \wedge \underline{a}_n),$$

[redacted]

where X1,..., Xm are the variables occurring in p. A database D satisfies p, denoted D \= p, if for each homomorphism h, {/?(ai),..., h(an)} C D implies h(a0) G D. The conjunction of the first-order sentences associated to the rules of a program P is denoted SP .A model of P is a database for SCH (P) that satisfies SP. The semantics of P on an input database D, denoted P(D), is the unique C- minimal model of P containing D. Given an n-ary CQ q : p(X) ^ p(X, Y) over SCH(P), the answer to q w.r.t. P and D is the set of n-tuples {t | t G q(P(D))}.

[redacted]

Example 4. Consider the program Pgraph given in Example 3, and the database

D = {edge(v1, V3), edge(v2, V3), edge(V3, V4), edge(v4,vs), edge(v5,^3), spjnode{v 1)}

for the schema EDB (Pgraph); the graph G encoded by D is depicted in Figure 1, where the special node is shaded. The BCQ q over SCH(P)

ans() ^ edge(X,Y), edge(Y,Z), edge(Z,X),
sp_reachable(X), sp-reachable(Y), sp.reachable(Z),

$$D = \{edge(v_1, v_3), edge(v_2, v_3), edge(v_3, v_4), edge(v_4, v_5), edge(v_5, v_3), sp\_node(v_1)\}$$

[redacted]

$$ans() \leftarrow edge(X, Y), edge(Y, Z), edge(Z, X), sp\_reachable(X), sp\_reachable(Y), sp\_reachable(Z),$$

asks whether a directed triangle in G is reachable from the set of special nodes of G. It is easy to verify that Pgraph(D) = D U |^J {sp.reachable(v*)}. ie{1,3,4,5}

$$P_{graph}(D) = D \cup \bigcup_{i \in \{1,3,4,5\}} \{sp\_reachable(v_i)\}.$$

It is straightforward to see that there exists a homomorphism that maps body(q) to Pgraph {D): ^lld tllUS Pgraph (D) |= q. ■

It is important to say that for a given program P and a database D for EDB (P), there exists always a unique C-minimal model of SP containing D, i.e., P(D) exists. Moreover, it is always possible to construct it. In particular, P(D) is the unique database for SCH(P) which satisfies the following conditions:

— P{D) = {o | for each model M of P containing D,a G M},

— dom(P(D)) C dom(P) U dom(D),

— for each r G EDB(P), r(P(D)) = r(D).

The above conditions actually provide an algorithm for computing the semantics of Datalog programs. However, this is clearly an inefficient procedure. A more reasonable algorithm is provided by the fixpoint semantics.

Fixpoint Semantics. The fixpoint semantics of Datalog programs relies on an operator called the immediate consequence operator. In fact, this operator produces new facts starting from known facts. It is possible to show that, given a Datalog program P and a database D, P(D) can be defined as the smallest solution of a fixpoint equation involving that operator. This approach can be seen

as an implementation of the model-theoretic semantics.

Let P be a Datalog program and D a database for SCH(P). A fact a is an immediate consequence for D and P if either a G r(D) for some predicate r G EDP(P), or there exists a rule a0 <— a1;... ,an in P and a homomorphism h such that h({a1;... ,an}) C D and a = h(a0). The immediate consequence operator of P, denoted TP, is the mapping from databases for SCH(P) to databases for SCH(P) defined as follows: for each database D, Tp(D) = {a | a is an immediate consequence for D and P}. We write Tp(D) for the result obtained by applying i times TP starting from D. Formally, T0(D) = D and Tp+1(D) = TP(Tp(D)); let Tp(D) = Ji>0 TP(D). From the fact that D C TP(D) and the monotonicity of TP, it is clear that Tp(D) C Tp+1(D), for each i ^ 0. It is well-known that for a Datalog program P and a database D for EDB(P), Tp (D) is the minimum fixpoint of TP containing D which coincides with P(D). Interestingly, Tp(D) - and thus P(D) - can be obtained by applying finitely many times TP. More precisely, given a Datalog program P, for each database D there exists an integer kD (which depends on D) such that Tp(D) = Tp,D (D) = P(D). The smallest such integer is called the stage for D and P, denoted stage(D,P).

Example 5. Recall the program Pgraph given in Example 3, and the database D given in Example 4. Clearly,

Tpsraph(D) = DU {sp.reachable(vi)}

TParapSD) = TP9raAD) U Up-

[redacted]

[redacted]

reachable(v3)}
TPSruPh^D) = TPSr*ph(D) U {sp-reachable(v4)}
TPSruPh^D) = TP9ruPh^D) U {sp-reachable(v5)}
ThpPh (D)= TPSaPhh (D).

$$T_{P_{graph}}(D) = D \cup \{sp\_reachable(v_1)\}$$
$$T^2_{P_{graph}}(D) = T_{P_{graph}}(D) \cup \{sp\_reachable(v_3)\}$$
$$T^3_{P_{graph}}(D) = T^2_{P_{graph}}(D) \cup \{sp\_reachable(v_4)\}$$
$$T^4_{P_{graph}}(D) = T^3_{P_{graph}}(D) \cup \{sp\_reachable(v_5)\}$$
$$T^5_{P_{graph}}(D) = T^4_{P_{graph}}(D).$$

[redacted]

Hence, Tp(D) = Tp(D). Notice that Pgraph{D) = Tp(D). m

[redacted]

3.3 Complexity of Datalog

We conclude this section by discussing the complexity of query answering under Datalog programs. Formally, given a Datalog program P, a database D for EDB(P), an n-ary CQ q over SCH(P), and an n-tuple t G rn, CQAns is defined as the problem of deciding whether t G q(P(D)). In case that q is a BCQ (and thus t is the empty tuple), the above problem is called BCQAns. Following Vardi's taxonomy [37], the data complexity of the above problems is the complexity calculated taking only the database as input, while the Datalog program and the query are fixed. The combined complexity is the complexity calculated considering as input, together with the database, also the program and the query. It is well-known that CQAns and BCQAns are LOGSPACE-equivalent; this result is implicit in [15], and stated explicitly in [8]. Henceforth, for brevity, we focus only on BCQAns, and all complexity results carry over to CQAns.

Given a Datalog program P and a database for EDB (P), the total number of atoms that can appear in P(D) is bounded by
ISCH(P)| • (Idom(D) U dom(P)|)w ,

[redacted]

$$|SCH(P)| \cdot (|dom(D) \cup dom(P)|)^w,$$

where w is the maximum arity over all predicates of SCH (P). Clearly, if the program is fixed (resp., non-fixed), then P(D) can be constructed by applying polynomially (resp., exponentially) many times the operator TP, while each application is feasible in polynomial (resp., exponential) time. This implies that query answering under Datalog programs is in polynomial time in data complexity, and in exponential time in combined complexity.

Interestingly, the above upper bounds are also lower bounds. As shown in [18], the PTlME-hardness can be established by constructing a simple Datalog metainterpreter for the class of propositional Logic Programs whose clauses have at most three body-atoms, denoted LP(3) . More precisely, given an LP(3) program P, a database DP can be constructed in LOGSPACE as follows: each rule a is encoded by an atom t(a), while each rule a0 •<— a1;... ,ai is encoded by an atom rj(a0,... , aj, where r* is an (i + l)-ary predicate. Then, P which is actually stored as described above in DP can be evaluated by a fixed Datalog program PMI constructed as follows: for each i G [3], add a rule

t(Xo) ^ t(X1),...,t(Xi),ri(Xo,...,Xi).

Intuitively, t(b) means that the propositional atom b is true. It is not difficult to see that, given a propositional atom a, P \= a iff PMI(Dp) \= q, where q is the BCQ p t(a). The EXPTlME-hardness can be established by simulating the behavior of an exponential time Turing machine by means of a Datalog program. In fact, this holds

[redacted]

[redacted]

$$t(X_0) \leftarrow t(X_1), \ldots, t(X_i), r_i(X_0, \ldots, X_i).$$

[redacted]

even if the input database is empty, and only the constants 0 and 1 are allowed in the program; for details we refer the reader to [18].

From the above discussion we immediately get the following complexity char-acterization of query answering under Datalog programs.

Theorem 1. BCQAns under Datalog programs is PTIME-complete in data com-plexity, and EXPTIME-complete in combined complexity.

An interesting (semantic) property of Datalog programs is boundedness. A Dat- alog program is bounded if there exists a constant k such that, for each database for EDB(P), stage(D,P) ^ k; recall that in general stage(D,P) depends on D. If a program is bounded, then it is actually non-recursive, even if syntactically appears to be recursive. It is well-known that every bounded Datalog program can be transformed into an equivalent non-recursive program. For example, the (recursive) program

buys(X,Y) ^ trendy(X), buys(Z,Y)
buys (X,Y) ^ likes (X, Y)

is bounded, and the atom buys (Z, Y) in the body of the first rule can be replaced by likes (Z,Y).

Given a bounded Datalog program P and a BCQ q : p ^ ¥>(X) over SCH(P), which can be seen as a Datalog rule, the program P U{p ^ ^(X)} is still bounded since p G SCH (P). This observation implies that a non-recursive Datalog program Pq can be constructed such that P(D) = q iff p G Pq(D), for every database D for EDP (P). The problem of deciding whether the propositional

$$buys(X,Y) \leftarrow trendy(X), buys(Z,Y)$$
$$buys(X,Y) \leftarrow likes(X,Y)$$

atom p belongs to Pq (D) can be easily reduced to the problem of evaluating a positive existential first-order sentence, obtained by "unfolding" (trải ra) the intensional predicates of Pq, over D; for details see [1]. For example, if Pq is the program

p ^ s(X),r(X,Y),t(Y,Z) r(X,Y) ^ t(Y,X),u(X),v(Y,Z) r(X,Y) ^ t(X,Y),

then the first-order sentence that must be evaluated is

3X3Y3Zs(X) A ((3Z' t(Y, X) A u(X) A v(Y,Z')) V t(X, Y)) A t(Y,Z).

Since evaluation of first-order sentences is feasible in AC0 in data complexity [40], the next result follows immediately.

Theorem 2. BCQAns under bounded Datalog programs is in ACo in data complexity.

As said, boundedness is a semantic (and not a syntactic) property. In fact, the problem of deciding whether a Datalog program is bounded is undecidable (see, e.g., [24]). This implies that only sufficient syntactic conditions, which will not detect boundedness in some cases, can be defined. Decidability results for particular subclasses are established, e.g., in [17,38].

4 Datalog Extensions

Datalog was designed and intended for reasoning over finite databases, assuming that only the values explicitly mentioned in the extensional database exist. For ontological reasoning, however, it would be desirable that an extended version of Datalog could be able to express the existence of certain values that are not necessarily from the EDB domain. This can be

$$\exists X \exists Y \exists Z\, s(X) \wedge ((\exists Z'\, t(Y,X) \wedge u(X) \wedge v(Y,Z')) \vee t(X,Y)) \wedge t(Y,Z).$$

achieved by allowing existentially quantified variables in rule heads. Other useful features for representing ontologies, which are not expressible by Datalog rules, are functionality and disjointness constraints. Such assertions can be supported by allowing the equality predicate and the truth constant false to appear in rule heads.

Example 6. As said, ontology languages are based on DLs. Such formalisms can express, for instance, that every person has exactly one father/mother who, moreover, is himself/herself a person, and also that fathers and mothers are disjoint sets, by the following axioms: Person C 3Father, Person C 3Mother, 3Father~ C Person, 3Mother~ C Person, 3Father~ disj 3Mother~, (funct Father) and (funct Mother). In an appropriate extended version of Datalog, the same can be expressed as:

3Y father (X, Y) ^ person(X)
3Y mother(X,Y) ^ person(X)
person(Y) ^ father (X, Y) person(Y) ^ mother(X,Y)
L ^ father(X, Y), mother(Z, Y)
Y = Z ^ father (X, Y), father (X, Z).

Observe that all the predicates occurring in the above program appear both in the body and in the head of some rule. Therefore, in the desired extended version of Datalog, we no longer require the distinction between extensional and intensional predicates. ■

Recently a family of Datalog-based languages, called Datalog±, which is a new framework for tractable ontology querying, has been introduced [10]. The precise aim of

[redacted]

$$\exists Y\, father(X,Y) \leftarrow person(X)$$
$$\exists Y\, mother(X,Y) \leftarrow person(X)$$
$$person(Y) \leftarrow father(X,Y)$$
$$person(Y) \leftarrow mother(X,Y)$$
$$\bot \leftarrow father(X,Y), mother(Z,Y)$$
$$Y = Z \leftarrow father(X,Y), father(X,Z).$$

[redacted]

[redacted]

Datalog± is to extend Datalog with existential quantification, the equality predicate, and the truth constant false, while preserving, not only decidability, but also tractability of query answering in data complexity. In view of the fact that query answering under Datalog extended with existential quantification is already undecidable [7], certain syntactic restrictions were needed (hence the symbol ±).

4.1 Syntax of Datalog3,=,x

A Datalog3,=,± rule p is an expression of the form

h au ..., an,

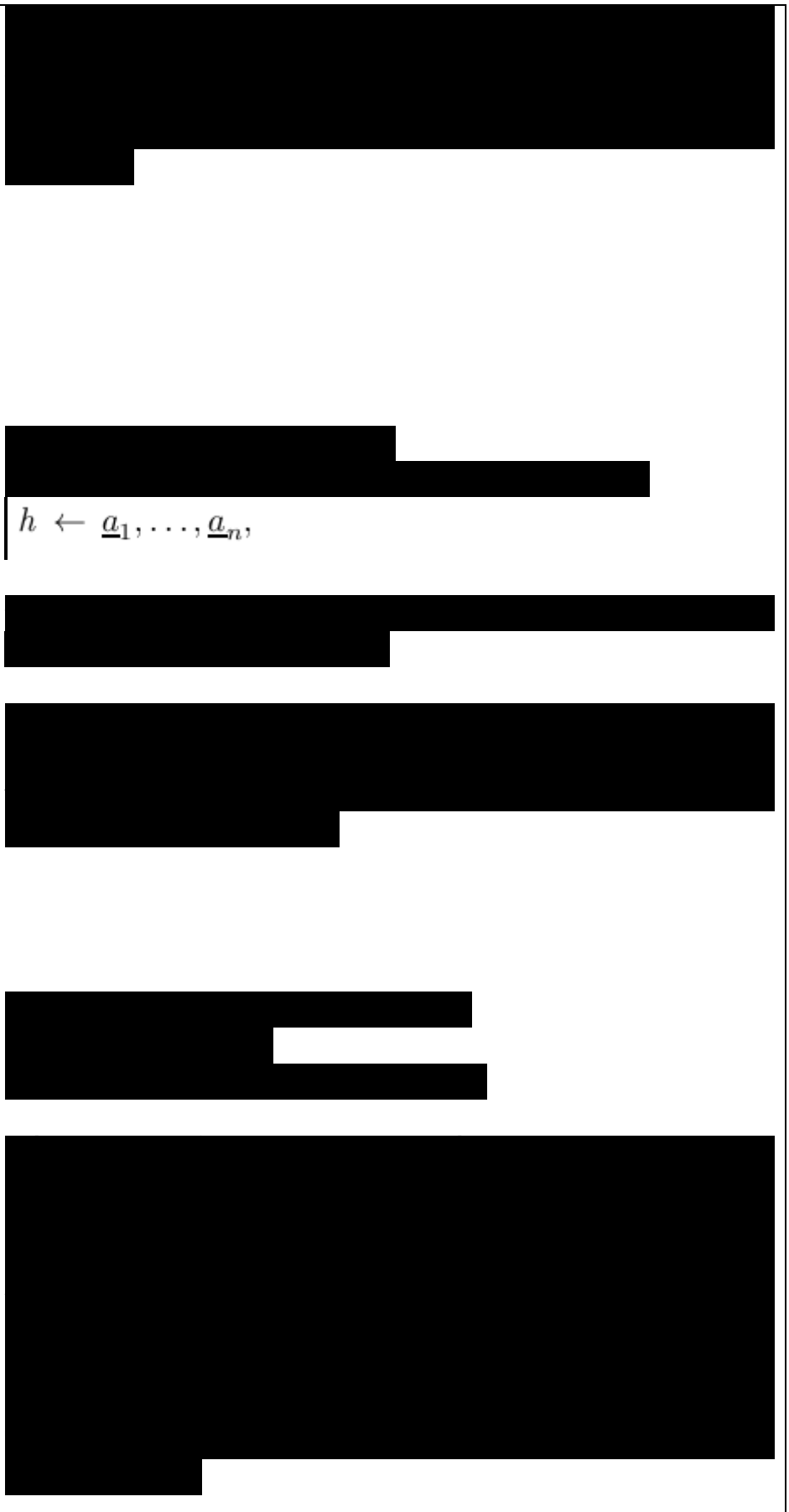
where a1;... ,an are atoms over a relational schema which contain constants of r and variables of rV, and h is either

— an expression 3Yi ... 3Ym a0, where to ^ 0, {Yi,..., Ymj C IV, {^l, • • •, Ynj and var({a1;... ,an}) are disjoint, and a0 is an atom which contains constants of r and variables of ({Yi,..., Ymj U var({a1;..., o,,})), or

— an expression X = Y, where {X, Y} C var({a1;... ,an}), or

— the symbol L which denotes the truth constant false.

If h is either an equality among variables or the symbol L, then n ^ 1; otherwise, n ^ 0. The expression h is called the head, denoted head(p), while the set of atoms {a-L,... ,an} is called the body, denoted body(p). A Datalog3^'1- program P is a finite set of Dataloga,=,± rules. The schema of P, written SCH(P), is the set of predicates occurring in P. Notice that SCH (P) is not partitioned, as in plain Datalog programs, into extensional and



intensional predicates. Obviously, the program given in Example 6 is a Datalog3,=,± program.

4.2 Semantics of Datalog3,=,x

A Model-Theoretic Approach. To define the semantics of Datalog3,=,± we follow an approach similar to that of the model-theoretic semantics of Datalog, i.e., we consider the given program as a first-order theory. More precisely, to a rule p : h a1;..., an, where var(body(p)) = {Xi,..., Xj.}, if h is of the form 3Yi ... 3Ym a0, then we associate the first-order sentence VXi... VXfc3Yi... 3Ym (a0 <— a1 A ... A an).

An instance I satisfies p if for each homomorphism /x, {p(ai), ■ ■ ., p(an)} Q I implies that there exists an extension pi of p, i.e., pi D p, such that p'(a0) G /. Sentences of the above form are known as tuple-generating dependencies (TGDs) in the database literature - essentially they say that the presence of some tuples in the instance implies the presence of some other tuple in the same instance. Notice that in general the head of a TGD can be a conjunction of atoms, and not just a single atom. In this way, however, we do not extend the expressive power of the language, since a set of TGDs can be reduced in LOGSPACE into an equivalent set of TGDs with a single atom as a head [8]. Now, if h is of the form X = Y, where {X, Y} C {X1,..., Xk}, then we associate to p the sentence VXi...VXfc (X = Y A...A aj.

An instance I satisfies p if for each homomorphism p, {p(a1),..., p(an)} C I implies p(X) = p(Y). Sentences of the above form are known as

[redacted]

$$\forall X_1 \ldots \forall X_k \exists Y_1 \ldots \exists Y_m \, (\underline{a}_0 \leftarrow \underline{a}_1 \wedge \ldots \wedge \underline{a}_n).$$

[redacted]

$$\forall X_1 \ldots \forall X_k \, (X = Y \leftarrow \underline{a}_1 \wedge \ldots \wedge \underline{a}_n).$$

[redacted]

equality- generating dependencies in the database literature - basically they assert that the presence of some tuples in the instance implies that certain tuple components are equal. Finally, if h = ±, then we associate to p the sentence

\X\ .. .yXk (_L i— <2-^ A ... A fln).

An instance I satisfies p if for each homomorphism p, {p(a1),..., p(an)} % I. Such sentences are known as negative constraints, and actually they forbid the existence of some tuples in the instance.

The conjunction of the first-order sentences associated to the rules of a Datalog3,=,± program P is denoted SP. A model of P is an instance for SCH(P) that satisfies SP. The semantics of P on an input database D, denoted P(D), is an instance M for SCH(P) such that M D D, M is a model of P, and for each model M' of P containing D, there exists a homomorphism p such that p(M) C M'; such an instance is called universal model of P w.r.t. D. Intuitively speaking, a universal model it contains no more and no less information than what the given program requires. It is not difficult to verify that, in general, there exist more than one universal models of P w.r.t. D.

However, by definition, they are the same up to homomorphic equivalence, i.e., for each pair of universal models M1 and M2, there exist homomorphisms h1 and h2 such that h1(M1) C M2 and h2(M2) C M1. Thus, P(D) us unique up to homomorphic equivalence. Given an n-ary CQ q : p(X) ^ ¥>(X, Y) over SCH(P), the answer to q w.r.t. P and

$$\forall X_1 \ldots \forall X_k \, (\bot \leftarrow \underline{a}_1 \wedge \ldots \wedge \underline{a}_n).$$

D is the set of n-tuples {t | t G q(P(D))}. It is important to say that for query answering, homomorphically equivalent instances are indistinguishable, i.e., given two instances I and I' which are the same up to homomorphic equivalence, q(I) and q(I') coincide. From the above discussion we conclude that, for query answering purposes, P(D) is defined uniquely.

A Fixpoint Approach for Datalog3. Interestingly, analogous to the fix-point semantics of Datalog programs, given a Datalog3 program P - in the rest of the paper, we denote Datalog3 the formalism obtained by allowing only existential quantification in rule heads - and an input database D, P(D) can be (constructively) defined as the least fixpoint of a monotonic operator (modulo homomorphic equivalence). This can be achieved by exploiting the well-known chase procedure, introduced for checking implication of dependencies [31], and later for checking query containment [28]. The chase is an algorithm that, roughly speaking, executes the rules of P starting from D in a forward chaining manner by inferring new atoms, and inventing new null values whenever an existential quantifier needs to be satisfied. The formal definition follows.

Consider a Datalog3 rule p : 3Yi ... 3 Ym a0 <— a-^,... ,an, and an instance I. We say that p is applicable to I if there exists a homomorphism h such that h({ai,..., an}) C /, but there is no h' D h such that h(a0) G I. Let I' be the instance I U {^'(cio)}, where h' D h is such that

h!{Yi) is a fresh labeled null of rN not occurring in I, for each i G [m], and h'(Yi) = h'(Yj), for each pair (i,j) of distinct integers of [m]. We say that the result of applying p to I with h is I',
and write I I'; in fact, I I' defines one single chase step. Consider now a Datalog3 program P, and an input database D. A chase sequence of D w.r.t. P is a sequence of chase steps Ii Ii+1, where i ^ 0, I0 = D, and pi G P. The result of the chase of D w.r.t. P, denoted chase(D,P), is defined as follows:

— A finite chase of D w.r.t. P is a finite chase sequence Ii Ii+1, where 0 ^ i < m, and there is no p G P which is applicable to Im; let chase (D, P) = Im.

— An infinite chase sequence Ii Ii+1, where i ^ 0, is fair (công bằng, hợp lí) if whenever a rule p G P is applicable to Ii with homomorphism h, then there exist h' D h and k > i such that h'(head(p)) G Ik. An infinite chase of D w.r.t P is a fair infinite chase sequence Ii Ii+1, where i ^ 0; let chase(D,P) = |Ji=0 Ii.

A useful technical notion is the depth of the chase. Roughly, the lower the depth of an atom, the earlier the atom has been obtained during the construction of chase(D, P). For an atom a G D, let depth(a) = 0, and for an atom a G chase(D, P) obtained during the chase application Ii Ii+i, let depth(a) = max.f>efl(body(p)){depth(b)} + 1. The chase of D w.r.t P up to depth k ^ 0, denoted chasek{D, P), is defined as the instance {a \ a G chase(D, P) and depth(a) ^ £;}.

Example 7. Consider the Datalog3

program P:
P1 : 3Y father(X, Y) ^ person(X) p2 : person(Y) ^ father(X, Y),
which is a subset of the program given in Example 6, and the input database D = {person(john)}. A fair infinite chase of D w.r.t. P follows:
D
D U {father (john, zi)}
l} D U {father(john,z1), person(z1)}
D U {father(john, z1), person(z1), father(z1, z2)}
D U {father(john, z1)} U (Jjj'^person(zj), father(zj, zJ+1)}
where z1,z2,... are labeled nulls of rN. Thus, chase(D, P) is the infinite instance D U {father(john, Z1), person(Z1)} U |J°=1 {father(zP,zP+1), person(z m)}- ■
It turns out that the result of the chase of an input database D w.r.t. a Datalog3 program P is a universal model of P w.r.t. D (see, e.g., [19,23]). This can be shown by induction on the number of applications of the chase step. In fact, it is possible to show that, for each k ^ 0, there exists a homomorphism hk that maps the initial part of chase(D,P) obtained after k chase step applications to each model of P containing D. Moreover, it holds that h0 C ... C hk C ..., and therefore the homomorphism |J°=0 hP maps chase (D, P) to each model of P containing D. Since, by definition, chase(D,P) is a model of P containing D, we conclude that chase(D, P) is a universal model of P w.r.t. D.
It is easy to verify that the result of the chase of D w.r.t P is not unique since it depends on the order that the rules of P are executed. In other

$\rho_1 : \exists Y\, father(X,Y) \leftarrow person(X) \qquad \rho_2 : person(Y) \leftarrow father(X,Y),$

[redacted]

$$
\begin{array}{ll}
 & D \\
\xrightarrow{\rho_1, h_1=\{X\to john\}} & D \cup \{father(john, z_1)\} \\
\xrightarrow{\rho_2, h_2=\{X\to john, Y\to z_1\}} & D \cup \{father(john, z_1), person(z_1)\} \\
\xrightarrow{\rho_1, h_3=\{X\to z_1\}} & D \cup \{father(john, z_1), person(z_1), father(z_1, z_2)\} \\
\vdots & \\
\xrightarrow{\rho_1, h_{2i-1}=\{X\to z_{i-1}\}} & D \cup \{father(john, z_1)\} \cup \bigcup_{j=1}^{i-1} \{person(z_j), father(z_j, z_{j+1})\} \\
\vdots &
\end{array}
$$

[redacted]

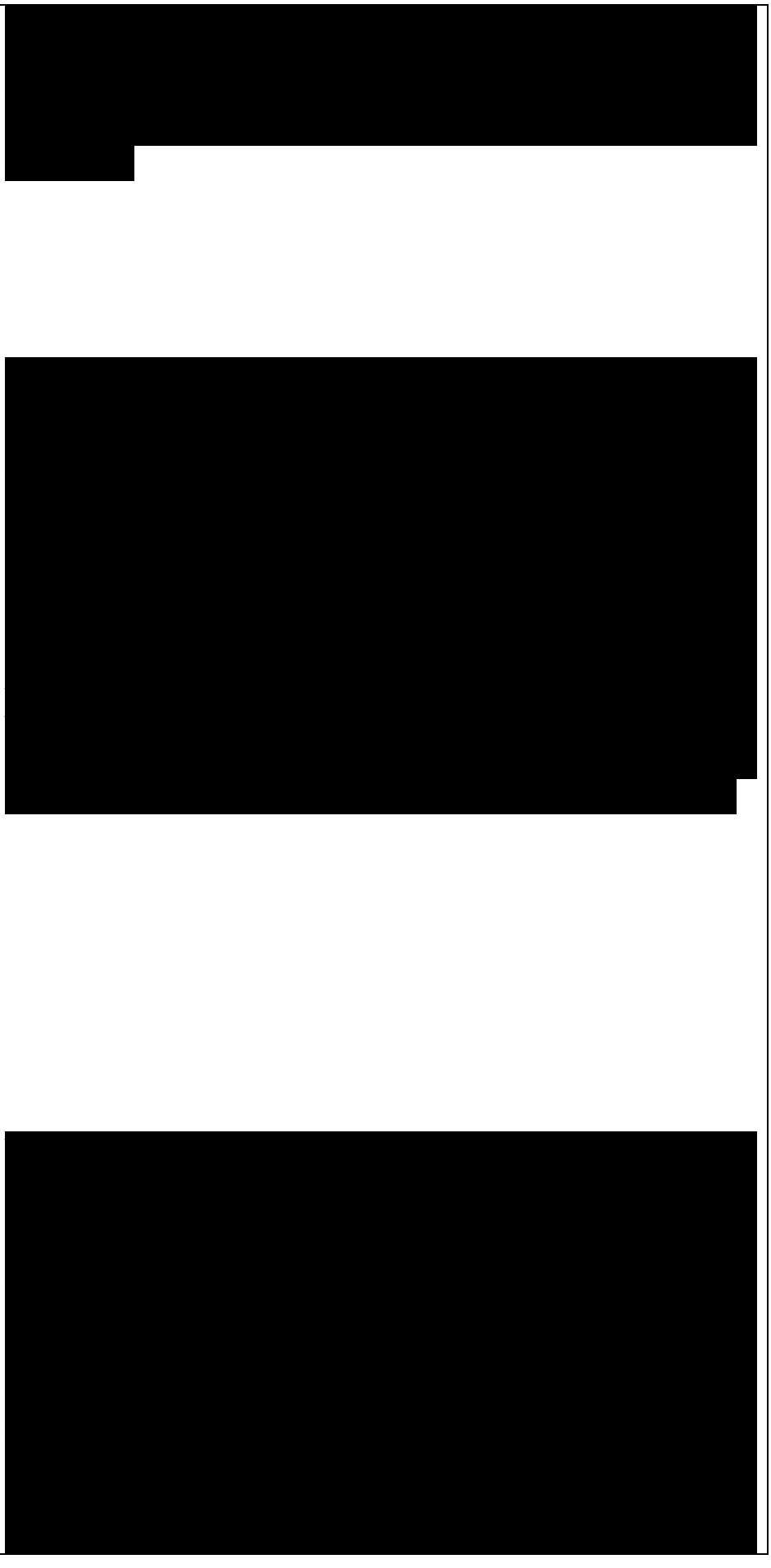
[redacted]

[redacted]

words, different chase sequences may yield different results. However, each result is a universal model of P w.r.t. D, and thus chase(D,P) is unique up to homomorphic equivalence. From the above discussion we conclude that P(D) and chase(D,P) coincide (up to homomorphic equivalence).

The Challenge of Infinity. Recall that for a Datalog program P and an input database D, P(D) is finite and it is always possible to construct it. In fact, the fixpoint semantics of Datalog programs provide an efficient (w.r.t. the size of the data) algorithm, based on the immediate consequence operator, which constructs P(D). Unfortunately, the situation changes dramatically if P is a Datalog3 (and thus, a Datalog3,=,±) program. Due to the existentially quantified variables in rule heads, P(D) is in general infinite, and thus not explicitly computable. As illustrated in Example 7, the result of the chase procedure is in general infinite even for very simple Datalog3 programs, and for extremely small input databases.

Already for Datalog3 query answering is undecidable [7]. Worse than that, undecidability holds even if both the program and the query are fixed, and only the database is given as input [8]. It is thus necessary to identify expressive fragments of Datalog3,=,± for which query answering is decidable, and also tractable in data complexity. In what follows we present such fragments of Datalog3,=,±. First, in Subsection 4.3, we focus on Datalog3, and we



discuss how we can regain decidability, and also tractability in data complexity, of query answering. Then, in Subsection 4.5, we discuss how = can be safely added, and also that L can be easily treated.

4.3 Guarded Datalog3

Guardedness, proposed by Andreka et al. [2], is a well-known restriction of first- order logic that ensures decidability of satisfiability, i.e., the problem of deciding whether a first-order theory has at least one model. Inspired by the guarded fragment of first-order logic, guarded Datalog3 has been proposed recently by Cali et al. [8,9]. A guarded Datalog3 rule p is a Datalog3 rule of the form h a1,..., an, where at least one atom a G body(p) contains all the variables occurring in body(p), i.e., var(a) = var(body(p)). The rightmost such atom is called the guard of p. A guarded Datalog3 program P is a finite set of guarded Datalog3 rules. The sentences of the first-order theory SP are known as guarded TGDs [8,9]. Notice that SP falls in the guarded fragment of first-order logic.

The decidability of BCQAns follows from the fact that the result of the chase of a database w.r.t. to a guarded Datalog3 program has finite treewidth, i.e., is a treelike structure; for more details we refer the interested reader to [8]. The data complexity of BCQAns under guarded Datalog3 programs was investigated in [9]; in fact, was shown that it is PTlME-complete. Let us briefly discuss how this results was obtained.

First, was established that guarded

Datalog3 enjoys the so-called bounded guard-depth property (BGDP). The chase graph of a database D w.r.t. a guarded Datalog3 program P is a labeled directed graph (V, E, X), where V is the node set, E is the edge set, and X is a node labeling function V — chase(D,P). For each atom a G D, there exists a node v G V such that A(v) = a. For each atom a G chase(D,P) obtained during a chase step /j Ii+i, where p is of the form 3Yi ... 3Ym a0 <— a-^,..., an and the guard of p is a •, there exist edges (v\, u),..., (vn, u), where Ai(wj) = and Ai(w) = h(a0) = a; the node Vj is marked as guard. The guarded chase forest of D w.r.t. P is the restriction of the chase graph of D w.r.t. P to all nodes marked as guards and their children.

Example 8. Consider the guarded Datalog3 program P:

p1 : 3Zr(Z,X) ^ r(X,Y),s(Y) p2 : s(X) ^ r(X,Y),

and the database D = {r(a,b), s(b)}. An initial part of the chase graph (resp., guarded chase forest) of D w.r.t. P is depicted in Figure 2(a) (resp., 2(b)). The

r(a,b)° s(bf r(a,b)° s(bf

(a) (b)

Fig. 2. Chase graph and guarded chase forest for Example 8

edges are labeled by the applied rules, and the number on the upper right side of each atom indicates its depth; formally not part of the graph (resp., forest). ■

The BGDP guarantees the following: whenever the given BCQ q is entailed by chase(D,P), and thus

$\rho_1 : \exists Z\, r(Z, X) \leftarrow r(X, Y), s(Y) \qquad \rho_2 : s(X) \leftarrow r(X, Y),$

P(D) = q, then the whole chase derivation of the query atoms is contained in a finite, initial part C of the guarded chase forest of D w.r.t P, whose depth depends only the query and the program, but not on the database. Thus, the BGDP allows us to construct the finite part C of the guarded chase forest (instead of considering the infinite instance chase(D,P)), and then evaluate q over C. Finally, was shown that the construction of C can be done in polynomial time in the size of the data, and the PTIME upperbound of BCQAns under guarded Datalog3 programs in data complexity follows.

The PTlME-hardness of BCQAns under guarded Datalog3 in data complexity follows immediately from the fact that the single rule used in the Datalog metainterpreter for LP(3) (see Subsection 3.3) is actually a guarded Datalog3 rule (without existentially quantified variables).

The combined complexity of BCQAns under guarded Datalog3 has been investigated by Cali et al. in [8]; in fact, it is 2EXPTlME-complete.

Theorem 3. BCQAns under guarded Datalog3 programs is PTIME-complete in data complexity, and 2EXPTIME-complete in combined complexity.

Guarded Datalog3 strictly captures the DL ECHI, that is, the extension of EC with role hierarchies and inverse roles [4], without losing tractability of BCQAns in data complexity. In particular, in a normalized ECHI ontology we can have axioms of the form: A Q B, A n B Q C, A Q BE, A Q BE.B, BE Q A,

BE.A Q B and E Q S, where A, B, C are atomic concepts, and E, S are basic roles. It is easy to verify that DL axioms of the above form can be translated into guarded Datalog3 rules.

4.4 Linear Datalog3

Interestingly, a (semantic) property analogous to boundedness of Datalog programs can be defined for Datalog3 programs. A Datalog3 program P has the bounded derivation-depth property (BDDP) if for every input database D, and for every BCQ q, P(D) = q implies chasek (D, P) = q, where k depends only on P and q. As shown in [9], the problem of deciding whether chasek (D, P) = q can be reduced to the problem of evaluating a first-order query Pq over D. Roughly, Pp can be constructed as follows. Suppose that P(D) = q which implies that chasek(D,P) = q. Hence, body(q) is mapped to chasek(D,P) via a homomorphism h. Since the derivation-depth and the number of body-atoms in rules of P are bounded, the cardinality of the set S of atoms at depth zero that are needed to entail the query is bounded. Thus, the number of all non-isomorphic sets S is also bounded. Consider the existentially quantified conjunction for every such set where q is answered positively. The first-order query Pq is the disjunction of all these formulas. Since evaluation of first-order queries is feasible in AC0 in data complexity [40], the next result follows.

Theorem 4. BCQAns under Datalog3 programs that have the BDDP is in

ACo in data complexity.

Recall that BDDP is a semantic property. The question that comes up is whether we can identify a syntactic fragment of Datalog3 programs that enjoys the aforementioned property. Let us first say that every class of Datalog3 programs under which query answering is PTIME-hard in data complexity does not have the BDDP. Assume that such a class C has the BDDP. By Theorem 4, we get that BCQAns under C is in AC0 in data complexity, and thus AC0 coincides with PTIME. But this is a contradiction since AC0 C ptime (see, e.g., [34]). Consequently, guarded Datalog3 does not have the BDDP. If we further restrict guarded Datalog3 by allowing only one body-atom in the rules (which is trivially a guard), then we get the class linear Datalog3 which enjoys the BDDP. The PSPACE-completeness of BCQAns (in combined complexity) under linear Datalog3 is implicit in the seminal work of Johnson and Klug [28], where the problem of conjunctive query containment under inclusion dependencies has been investigated. The next result follows.

Theorem 5. BCQAns under linear Datalog3 programs is in AC0 in data com-plexity, and PSPACE-complete in combined complexity.

4.5 Addition of = and ±

We now discuss how guarded and linear Datalog3 can be enriched with rules that have the equality predicate and the truth constant false in the head, without increasing the complexity. For simplicity and technical clarity, we consider only

[redacted]

[redacted]

[redacted]

[redacted]

$\rho : X = Y \leftarrow \underline{a}_1, \underline{a}_2,$

equality rules of the form
P • X == ^, (X<2■>
where and a2 have the same predicate, there are no constants in p, each variable of var(aj) occurs in ^ at most once, and each variable of var(a1) fl var(a2) occurs in both atoms and a2 at the same position. Notice that rules of this form, despite their simplicity, they are expressive enough to capture functionality constraints. For example, the rule Y = Z ^ father(X,Y),father(X, Z), given in Example 6, which asserts that each person has at most one father, is exactly of this form. In the sequel, we call guarded/linear Datalog3,= the formalism obtained by combining guarded/linear Datalog3 with equality rules of the above form.

It is known that query answering under linear Datalog3,= (and thus, also under guarded Datalog3,=) programs is undecidable; this is implicit in [16]. The reason for this is the interaction between existential and equality rules. Moreover, while the result of the chase procedure under Datalog3 programs is well-defined as the least fixpoint of a monotonic operator (modulo homomorphic equivalence) - see Subsection 4.2 - this is not true if we consider also equality rules. Despite the fact that the chase procedure under Datalog3,= programs can be formally defined, it is not clear how the result of an infinite chase involving both existential and equality rules should be defined; the sequence of sets obtained in such an infinite chase is, in general, neither monotonic nor convergent.

Separability. It would be useful to identify syntactic restrictions which guarantee that, for query answering purposes, the semantics of Datalog3,= programs coincide with the semantics of Datalog3, providing that the input database satisfies the equality rules. In other words, we need sufficient syntactic conditions for the property of separability. Consider a Datalog3,= program P, and let P3 (resp., P=) be the rules of P with an existential quantifier (resp., equality) in the head. P is separable if, for every input database D for SCH(P): if D = P=, then P(D) exists, and, for every BCQ q over SCH(P), P(D) = q iff P3(D) = q.

Separability allows us to answer queries under guarded/linear Datalog3,= programs by exploiting the algorithms for guarded/linear Datalog3. More precisely, given a program P, a database D for SCH(P), and a BCQ q for SCH(P), we can decide whether P(D) = q by applying the following algorithm: (1) if D = P=, then accept; (2) if P3D) = q, then accept; otherwise, reject. Interestingly, as shown in [9], the problem of deciding whether the input database satisfies P= it is not harder than conjunctive query evaluation. In fact, this problem is feasible in ACo if P= is fixed, and in NP in general. From the above discussion, we conclude that the notion of separability allows us to transfer, not only the decidability results, but also the upper complexity bounds derived for guarded/linear Datalog3.

Theorem 6. BCQAns under separable guarded/linear Datalog3,

programs has the same data/combined complexity as BCQAns under guarded/linear Datalog3 programs.

Non-conflicting Condition. Obviously, separability of Datalog3,= programs is a semantic (and not a syntactic) property. In what follows we present an efficiently checkable syntactic condition which is sufficient for separability of Datalog3,= programs. For brevity, given a rule p with an existential quantifier in the head, we define the set Up of universal positions of p as the set of positions in head(p) at which a variable of body (p) appears. Moreover, given an equality rule p, we define the set Jp of joined positions of p as the set of positions at which a variable that occurs in both atoms of body (p) appears; recall that we consider a special kind of equality rules with just two body-atoms. Formally, a Datalog3,= program P, where P3 (resp., P=) are the rules of P with an existential quantifier (resp., equality) in the head, is non-conflicting if, for each (p1,p2) G P3 x P=, the following holds: (i) UP1 ^ JP2, and (ii) if UP1 = JP2, then each existentially quantified variable in head(p1) occurs just once.

Example 9. Consider the Datalog3,= program P:

p1 : 3Y father (X, Y) ^ person(X) p2 : Y = Z ^ father (X, Y), father (X, Z),

which asserts that each person has exactly one father. Clearly, BP1 = {father [1]}, since the variable X is the first argument of head (p1), and JP2 = {father [1]} due to the variable X in body (p2). Since BP1 = JP2,

$\rho_1 : \exists Y\ father(X,Y) \leftarrow person(X) \quad \rho_2 : Y = Z \leftarrow father(X,Y), father(X,Z),$

and the existentially quantified variable Y occurs in head(pi) only once, we get that P is non-conflicting. ■

Given a non-conflicting Datalog3,= program P, and an input database D, if D = P=, then during the construction of chase (D,P3) it is not possible to violate an equality rule of P=. Let us explain briefly why this holds. Consider a pair (p1,p2) G P3 xP= such that the head-predicate of p1 and the predicate of the atoms of body (p2) is the same. In other words, the atom generated by applying p1 during the chase it is possible to violate p2. Since P is non-conflicting

— either JP2 \ UPl ^ 0, and the application of p\ generates an atom a with a "fresh" null at some position of JP2; hence, a does not violate p2,

— or UPl = JP2, and any newly generated atom a must have "fresh" distinct nulls (since each existentially quantified variable occurs in p1 just once) at all positions but those of JP2. Therefore, if a coincides with some existing atom in the chase at the positions of JP2, then a would not be added since p1 is not applicable; hence, again it is not possible to violate p2.

From the above informal discussion we conclude that chase(D, P3) is a universal model of P w.r.t. D. Consequently, if D = P=, then P(D) exists, and P(D) = q iff chase(D, P3) = q iff P3D) = q; hence, P is separable. The next result follows.

Theorem 7. BCQAns under non-conflicting guarded/linear Datalog3, pro-grams has the same data/combined complexity as

BCQAns under guarded/linear Datalog3 programs.

Adding L. We now show that the addition of L can be done effortless. Nonconflicting guarded/linear Datalog3,=,± is obtained by combining non-conflicting guarded/linear Datalog3,= with rules of the form _L a1,...,an. As shown in [9], the addition of this new feature does not increase the complexity of query answering. Let us explain why this holds. Consider a non-conflicting guarded/linear Datalog3,=,± program P, where P3 (resp., P=,P±) is the set of rules of P with existential quantifiers (resp., equality, the truth constant false) in the head, an input database D for SCH(P), and a BCQ q over SCH(P). For each rule p G P±, let qp be the BCQ p ^ body(p), where p G SCH(P). Also, let P3,= = P3 U P=. Clearly, if P3,= (D) = qp, for some p G P±, then there is no model of P w.r.t. D, and thus query answering is trivial since P(D) entails every BCQ over SCH(P). Consequently, P(D) = q iff P3,= (D) = q, or P3,= (D) = qp, for some p G P±. Since P is non-conflicting, we immediately get that P(D) = q iff P3(D) = q, or P3(D) = qp, for some p G P±. The next result follows.

Theorem 8. BCQAns under non-conflicting guarded/linear Datalog3, pro-grams has the same data/combined complexity as BCQAns under guarded/linear Datalog3 programs.

DL-Lite is a well-known family of DLs under which query answering is highly tractable, i.e., in AC0, in data complexity [13,36]. Notice that DL-

Lite forms the OWL 2 QL profile of OWL 2 DL . It is easy to verify that a DL-Litex, where X G {F, R, A}, ontology can be translated into a non-conflicting linear Datalog3,=,± program. Thus, non-conflicting linear Datalog3,=,± strictly captures the main languages of DL-Lite without losing the AC0 data complexity of BCQAns.

5 Further Reading

Several classes of Datalog3 programs have been investigated. Here are some examples. The class of weakly-guarded programs, an extension of guarded programs where the guard atom must cover only the body-variables that occur at affected positions, i.e., positions at which a null value can appear during the construction of the chase, has been investigated in [8]. The class of frontier-guarded programs, an extension of guarded programs where the guard atom must contain only the frontier, i.e., the set of variables that appear both in the body and the head, has been studied in [6]. As show in [9], guarded programs can be enriched with stratified negation, a simple non-monotonic form of negation often used in the context of plain Datalog. The class of weakly-acyclic programs has been proposed in the context of data exchange [23]. Notice that weak acyclicity guarantees the termination of the chase procedure. The more general class of super-weakly-acyclic programs has been studied in [32]. The class of sticky programs has been investigated in [11]. Stickiness is a decidability paradigm which

[redacted]

[redacted]

differs significantly from guardedness and weak acyclicity. The combination of stickiness with linearity and weak acyclicity is performed in [12]. The question whether weak acyclicity and guardedness can be combined in order to obtain more expressive decidable classes has been investigated in [29]. The work [22] presents the class of FDNC logic programs that allows for function symbols (F), disjunction (D), non-monotonic negation under the answer set semantics (N), and constraints (C), while standard reasoning tasks are decidable.	