Generating Plans from Proofs

Michael Benedikt, University of Oxford
and Balder ten Cate, LogicBlox and UC-Santa Cruz
and Efthymia Tsamoura, University of Oxford

Categories and Subject Descriptors: H.2.3 [**Information Systems**]: Database Management—*Languages*

General Terms: Theory; Languages

Additional Key Words and Phrases: access methods, optimization, hidden web

We present algorithms for answering queries making use of information about source integrity constraints, access restrictions, and access costs. Our method can exploit the integrity constraints to find plans even when there is no direct access to relations appearing in the query. We look at different kinds of plans, depending on the kind of relational operators that are permitted within their commands. To each type of plan we associate a semantic property that is necessary for having a plan of that type. The key idea of our method is to move from a search for a plan to a search for a proof of the corresponding semantic property, and then *generate a plan from a proof*. We provide algorithms for converting proofs to plans, and show that they will find a plan of the desired type whenever such a plan exists. We show that while discovery of one proof allows us to find a single plan that answers the query, we can explore alternative proofs to find lower-cost plans.

1. Introduction

This work concerns translating a *declarative source query* written in one vocabulary into a *target plan* that abides by certain *interface restrictions*. By a declarative source query we will always mean a fragment of first-order logic, or its equivalent in SQL. We will focus on queries given in the language of *conjunctive queries*, equivalent to SQL basic SELECT queries. By a plan we may mean a program that constructs query answers by interfacing with stored data.

What do we mean by an interface restriction on the target plan? The most basic kind of restriction is a *vocabulary-based restriction*, where the restriction is on the set of relations that are allowed to be referenced in the plan. We begin with a query Q written over relations $R_1 \dots R_j$, and want to convert it to query Q' making use of a different set of relations $V_1 \dots V_k$. Of course, if conjunctive queries Q and Q' mention different relations, Q can not be equivalent to Q' on arbitrary instances. But our schema will come with *integrity constraints* which restrict the possible instances of interest. We will thus be considering equivalence only on instances satisfying the constraints.

The most basic example of vocabulary-based restriction comes from *reformulating queries over views*. We have a collection of view relations $V_1 \dots V_k$, where each V_i is associated with a query Q_i over some other set of relations $R_1 \dots R_j$. Given a query Q over $R_1 \dots R_j$, the goal is to find a query Q' that mentions only $V_1 \dots V_k$ where Q' is equivalent to Q over all instances I for $R_1 \dots R_j$, where the instances are extended to $V_1 \dots V_k$ by interpreting each V_i by $Q_i(I)$. Such a Q' is called a *reformulation of Q over the views*. Generally, additional restrictions will be put on Q' —- e.g. it should be a conjunctive query, or a union of conjunctive queries, or definable in relational calculus. The view-based query reformulation problem can thus be seen as a special case of vocabulary-based restriction, where the integrity constraints are of the form

$$\forall x_1 \dots \forall x_n \; V_i(x_1 \dots x_n) \leftrightarrow Q_i(x_1 \dots x_n)$$

The information available via an interface may or may not be sufficient to answer a query. The reformulation problem in this case is to determine whether there is sufficient information, and if so to generate a query making use of the view predicates.

Example 1.1. Consider a database with a table Professor containing ids, last names, and departments of professors, as well as a table Student listing the id and last name of each students, as well as their advisor's id.

The database does not allow users to access the Professor and Student table directly, but instead exposes a view Professor′ where the id attribute is dropped, and a table Student′ where the advisor's id is replaced with the advisor's last name.

That is, Professor′ is a view defined by the query:

$$\{\mathsf{lname}, \mathsf{dname} \mid \exists \mathsf{profid}\ \mathsf{Professor}(\mathsf{profid}, \mathsf{lname}, \mathsf{dname})\}$$

or equivalently by the constraints:

$$\forall \mathsf{profid}\ \forall \mathsf{lname}\ \forall \mathsf{dname}\ \mathsf{Professor}(\mathsf{profid}, \mathsf{lname}, \mathsf{dname}) \rightarrow \mathsf{Professor}'(\mathsf{lname}, \mathsf{dname})$$
$$\forall \mathsf{lname}\ \forall \mathsf{dname}\ \mathsf{Professor}'(\mathsf{lname}, \mathsf{dname}) \rightarrow \exists \mathsf{profid}\ \mathsf{Professor}(\mathsf{profid}, \mathsf{lname}, \mathsf{dname})$$

Student′ is a view defined by the query:

$$\{\mathsf{studid}, \mathsf{lname}, \mathsf{profname} \mid$$
$$\exists \mathsf{profid}\ \exists \mathsf{dname}\ \mathsf{Student}(\mathsf{studid}, \mathsf{lname}, \mathsf{profid}) \wedge \mathsf{Professor}(\mathsf{profid}, \mathsf{profname}, \mathsf{dname})\}$$

or equivalently by constraints:

$$\forall \mathsf{studid}\ \forall \mathsf{lname}\ \forall \mathsf{dname}\ \forall \mathsf{profid}\ \forall \mathsf{profname}$$
$$\mathsf{Professor}(\mathsf{profid}, \mathsf{profname}, \mathsf{dname}) \wedge \mathsf{Student}(\mathsf{studid}, \mathsf{lname}, \mathsf{profid}) \rightarrow$$
$$\mathsf{Student}'(\mathsf{studid}, \mathsf{lname}, \mathsf{profname})$$
$$\forall \mathsf{studid}\ \forall \mathsf{lname}\ \forall \mathsf{profname}\ [\mathsf{Student}'(\mathsf{studid}, \mathsf{lname}, \mathsf{profname}) \rightarrow$$
$$\exists \mathsf{profid}\ \exists \mathsf{dname}\ \mathsf{Professor}(\mathsf{profid}, \mathsf{profname}, \mathsf{dname}) \wedge \mathsf{Student}(\mathsf{studid}, \mathsf{lname}, \mathsf{profid})]$$

Consider a query asking for the last names of all students that have an advisor in the history department. This query can *not* be answered using the information in the views, since knowing the advisor's name is not enough to identify the department.

On the other hand, the views are clearly sufficient to answer a query asking for the last names of students whose advisor has last name Jones, and we can reformulate that query as a selection over Student′ on profname "Jones".□

Naturally, constraints need not come from views. A natural use of constraints is to represent *relationships between sources*, such as overlap in the data. This overlap can be exploited to take a query that is specified over a source that a priori does not have sufficient data, and reformulate it over a source that provides the necessary data.

Example 1.2. We consider an example schema from [Onet 2013] with a relation Employee where a row contains an employee's id, the employee's name, and the id of the employee's department, and also a relation Department, with each row containing the department's id, the department's name, and the id of the department's manager.

The schema also contains the following two constraints:

$$\forall \mathsf{deptid}\ \forall \mathsf{dname}\ \forall \mathsf{mgrid}\ \mathsf{Department}(\mathsf{deptid}, \mathsf{dname}, \mathsf{mgrid}) \rightarrow \exists N\ \mathsf{Employee}(\mathsf{mgrid}, N, \mathsf{deptid})$$
$$\forall \mathsf{eid}\ \forall \mathsf{ename}\ \forall \mathsf{deptid}\ \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid}) \rightarrow \exists D \exists M\ \mathsf{Department}(\mathsf{deptid}, D, M)$$

That is, every department has a manager, and every employee works in a department. Suppose further that only the relation Department is accessible to a certain class of users. Intuitively, it should still be possible to answer some questions that one could ask concerning the relation Employee, making use of the accessible relation Department.

For example, suppose a user poses the query asking for all department ids of employees, writing it like this:

$$Q = \{\mathsf{deptid} \mid \exists \mathsf{eid}\ \exists \mathsf{ename}\ \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid})\}$$

Renaming the variables, the query can be reformulated as:

$$Q' = \{\mathsf{deptid} \mid \exists \mathsf{dname}\ \exists \mathsf{mgrid}\ \mathsf{Department}(\mathsf{deptid}, \mathsf{dname}, \mathsf{mgrid})\}$$

□

Access methods and binding patterns. We will look at a finer notion of interface based on *binding patterns*, which state that a relation can only be accessed via lookups where certain arguments must be given. The most obvious example is a relation that can only be accessed via an indexed lookup on a certain subset of the attributes. Another example of restricted interfaces that can be modeled using relations with binding patterns comes from web forms. Thinking of the form as exposing a virtual table, the mandatory fields must be filled in by the user, while submitting the form returns all tuples that match the entered values. A third example comes from web services, where the mandatory fields correspond to arguments of a function call.

Example 1.3. Consider a Profinfo table containing information about faculty, including their last names, office number, and id, but with a restricted interface that requires giving an id as an input. The query Q asking for ids of faculty named "Smith" cannot be answered over this schema. That is, there is no query over the schema that will return exactly the set of tuples satisfying Q.

But suppose another source has a Udirectory table containing the id and last name of every university employee, with an interface that allows one to access the entire contents of the table. Then we can reason that Q has a plan that answers it: a plan would pull tuples from the Udirectory table, select those corresponding to "Smith", and check them within the Profinfo table. □

In the above example, reasoning about access considerations was straightforward, but in the presence of more complex schemas we may have to chain several inferences, resulting in a plan that may make use of several auxiliary accesses.

Example 1.4. We consider two telephone directory datasources with overlapping information. One source exposes information from $\mathsf{Direct1}(\mathsf{uname}, \mathsf{addr}, \mathsf{uid})$ via an access requiring a uname and uid. There is also a table $\mathsf{Ids}(\mathsf{uid})$ with no access restriction, that makes available the set of uids (hence a referential constraint from Direct1 into Ids on uid). The other source exposes $\mathsf{Direct2}(\mathsf{uname}, \mathsf{addr}, \mathsf{phone})$, requiring a uname and addr, and also a table $\mathsf{Names}(\mathsf{uname})$ with no access restriction that reveals all unames in Direct2 (that is, a referential constraint from Direct2 to Names). There is also a referential constraint from Direct2 to Direct1 on uname and addr. Consider a query asking for all phone numbers in the second directory:

$Q = \{\mathsf{phone} \mid \exists\ \mathsf{uname}\ \exists \mathsf{addr}\ \mathsf{Direct2}(\mathsf{uname}, \mathsf{addr}, \mathsf{phone})\}$.

There is a plan that answers this query: it gets all the uids from Ids and unames from Names first, puts them into the access on Direct1, then uses the uname and addr of the resulting tuples to get the phone numbers in Direct2.□

We emphasize that our goal in this work is getting plans which give *complete answers to queries*. This means that if we have a query asking for the office number of all professors with last name "Smith", the plan produced should return all tuples in the answer, even if access to the Professor relation is limited.

We will look not just at getting any plan in the target language, but one with low cost. Examples of access cost include the cost in money of accessing certain services and the cost in time of accessing data through either web service calls, iteratively inputing into web forms, or using particular indices.

Plan-generation approach. The paper will overview a general approach that emerged from mathematical logic (starting with the work of William Craig [Craig 1957]) adapted to the database setting by Segoufin and Vianu [Segoufin and Vianu 2005]. The "meta-algorithm" for plan-generation is as follows:

(1) Isolate a *semantic property* that any input query Q must have with respect to the class of target plans and constraints Σ in order to have an equivalent plan of the desired type.
(2) Express this property as a *proof goal* (in the language we use later on, an *entailment*): a statement that formula φ_2 follows from φ_1.
(3) Search for a proof of the entailment, within a given proof system. Here we will focus on chase proofs, a well-known proof system within databases.
(4) From the proof, extract a plan.

We will show that this approach can be applied to a variety of restrictions on the plan, with different plan targets corresponding to different entailments. We prove a number of theorems saying that the method is complete: there is a plan exactly when there is a proof of the property. These completeness theorems give as a consequence a *definability or preservation theorem*: a query Q has a certain kind of plan iff it has a certain semantic property.

Thus our results will link: a semantic property of Q (with respect to $\mathcal{T}$ and Σ), the existence of a certain kind of plan equivalent to Q, and a proof goal such that from a proof we can generate the desired plan.

Adding on cost considerations. In the setting of overlapping datasources, there can be many plans with very distinct costs. Consider a variant of Example 1.3 in which there are two tables $\mathsf{Udirectory}_1$ and $\mathsf{Udirectory}_2$ that contain the necessary information. In this case we would have at least three plans: one that first accesses Udirectory1 as above and then checks the results in Profinfo, another that first accesses $\mathsf{Udirectory}_2$, and a third that accesses both $\mathsf{Udirectory}_1$ and $\mathsf{Udirectory}_2$ and intersects the results in middleware before doing the check in Profinfo. Which of these is best will depend on how costly access is to each of the directory tables, and what percentage of the tuples in the two directory tables match a result in Profinfo. Notice that these plans are not variants of one another, and one cannot be obtained from the other by applying algebraic transformations. We will present an algorithm that will find the lowest-cost plan for a class of cost functions on access plans. The main idea is to *explore the full space of proofs, but guiding the search by cost as well as proof structure.* Thus instead of generating a single proof and then sending the corresponding plan on for further optimization, we interleave exploration of proofs with calls to estimate cost (and perhaps further optimize) the corresponding plans.

Organization. We start by giving preliminaries on database schemas, query languages, and logics (Section 2), along with the plan language and associated query answering problems that we study in this work. Section 3 justifies the choice of plan language by showing its equivalence to other formalisms for defining queries that conform to access methods. Section 4 provides the proof goals that correspond to each kind of plan that we will be interested in, along with the corresponding semantic property. It then will present the main theorems of the paper, stating the equivalence of a semantic property, existence of a plan, and existence of a certain kind of proof.

The basic plan-generation algorithms that prove these theorems are presented in Section 5, which also shows that these algorithms always generate a correct plan from a proof. Coupled with earlier results from Section 4, this gives the main results on equivalence of proofs and plans in the paper.

In Section 6 we turn to getting plans with low cost. Section 7 gives conclusions, while 8 gives an overview of related work.

Acknowledgements. This work was funded by EPSRC EP/H017690/1 and EP/M005852/1, the Engineering and Physical Sciences Research Council UK. The authors are extremely grateful to the anonymous referees of TODS for their patient reading of the draft and numerous helpful suggestions.

2. Definitions

Logical notation. We will use standard terminology for describing queries in first-order logic, including the notion of free variable, quantifiers, connectives, etc. [Abiteboul et al. 1995].

If φ is a formula whose free variables include $\vec{x}$ and $\vec{t}$ is a sequence of constants and variables whose length matches $\vec{x}$ then $\varphi(\vec{x} := \vec{t})$ denotes the formula obtained by substituting each x_i with t_i. We will often omit universal quantifiers from formulas, particularly for formulas where the only quantifiers are universal. For example, we will write $P(x, y) \rightarrow Q(x, y)$ as a shorthand for $\forall x\, \forall y\, [P(x, y) \rightarrow Q(x, y)]$.

A relational schema contains a finite collection of *schema constants* and a finite collection of relations (or *tables*), each with an associated arity. We assume that distinct schema constants are associated with distinct value, and will identify the constant and the value. A database *instance* (or just instance) I for schema Sch assigns to every relation R in Sch a collection of tuples $I(R)$ of the right arity, in such a way that any integrity constraints of Sch are satisfied. We call $I(R)$ the *interpretation* of R in I. An association of a database relation R with a tuple $\vec{c}$ of the proper arity will be referred to as a *fact*. A database instance can equivalently be seen as a collection of facts. The *active domain* of an instance I is the union of the one-dimensional projections of all interpretations of relations: that is, all the elements that participate in some fact of I. When evaluating first-order formulas, we always assume the *active domain semantics* in which quantifiers range over the active domain of the instance.

In our plan-generation problems we have some "visible" information (e.g. a set of relations that our programs are allowed to access) and have to consider what underling instance is consistent with that. The notion of superinstance captures that an instance I' is consistent with the information provided by another instance I. If we have two instances I and I', and for every relation R, $I(R) \subseteq I'(R)$ we say that I is a *subinstance* of I', and I' is a *superinstance* of I.

Queries. By a query we mean a mapping from relation instances of some schema to instances of some other relation. A boolean query is a query where the output is a relation of arity 0. Since there are only two instances for a relation of arity 0, a boolean query is a mapping where the output takes one of two values, denoted True and False. Given a query Q and instance I, $Q(I)$ is the result of evaluating Q on I.

First-order logic sentences clearly can be used to define boolean queries: given an instance, the result of the query is True exactly when the sentence holds in the instance. Given a first-order logic formula with its free variables enumerated as $v_1 \ldots v_n$, we can associate a non-boolean query whose output relation has arity n, whose output on an instance I is the set of n-tuples of values $\vec{t}$ in the active domain of I for which the instance and the corresponding binding $v_1 := t_1 \ldots v_n := t_n$ satisfies the formula. For an instance I, formula φ and a binding bind for the variables of φ, we will write $I, \mathsf{bind} \models \varphi$ to mean that bind satisfies φ in I.

Relational algebra. An alternative notation for relations is to consider them as having named attributes. Relational algebra is a query language that references relations by name, using operations selection, projection, renaming, difference, and join, along with operator for each schema constant (taking no input and producing a single-attribute table whose sole cell contains that constant). The $USPJ$ fragment disallows the use of difference, while the $USPJ^{\neg}$ fragment allows the difference operator $E - E'$ to be applied only when $E' = E \ltimes_{\sigma} R$, where R is a relation symbol and σ is a set of equality conditions identifying each attribute of R with an attribute of E. In the $USPJ^{\neg}$ fragment we allow for the use of inequalities in selections and join conditions. The SPJ fragment further restricts $USPJ$ by not allowing union.

All of our fragments will, by convention, include the empty expression, denoted $\emptyset$, which always returns the empty set of tuples. We will freely move back-and-forth between logic-based notation and relational algebra notation, and also between positional and attribute-based notation for components of a tuple.

Relational algebra boolean queries are those which have no attributes as output. It is well-known that every relational algebra boolean query can be efficiently converted into an active-domain first-order logic formula and vice versa [Abiteboul et al. 1995].

Queries and constraints of particular interest. The problems we look at will generally have as inputs both a query and a set of constraints. For queries we will look at *conjunctive queries*, logical formula of the form $Q(\vec{x}) = \exists \vec{y}\ (A_1 \wedge \cdots \wedge A_n)$, where A_i is an atom using a relation of the schema, with arguments that are either variables from $\vec{x}$ and $\vec{y}$ or constants from the schema. Conjunctive queries are equivalent to queries defined in the SPJ fragment of relational algebra.

Existential formulas are those of the form $\exists x_1 \ldots x_n\ \varphi$, where φ is built up using the boolean operators. Existential formulas are equivalent in expressiveness to queries in the $USPJ^{\neg}$ fragment of relational algebra.

Although some of our results apply to constraints given by arbitrary first-order logic sentences, we will focus our attention on constraints given by *tuple-generating dependencies* (TGDs), given syntactically as

$$\forall \vec{x}\ [\varphi(\vec{x}) \rightarrow \exists \vec{y}\ \rho(\vec{x}, \vec{y})]$$

where φ and ρ are conjunctions of relational atoms, possibly including constants.

A special subclass consists of *Guarded TGDs* (GTGDs), in which φ is of the form $R(\vec{x}) \wedge \varphi'$ where $R(\vec{x})$ contains all variables of φ'. These in turn subsume *inclusion dependencies* (IDs): where φ and ρ are single atoms in which no variables are repeated and there are no constants. IDs are also called "referential constraints". An inclusion dependency with $\varphi = \forall \vec{x}\ \forall \vec{u}\ R(\vec{x}, \vec{u}) \rightarrow \exists \vec{y}\ S(\vec{x}, \vec{y})$ can also be written in the form $R[j_1 \ldots j_n] \rightarrow S[k_1 \ldots k_n]$, where j_i is the positions containing exported variable x_i in $R(\vec{x}, \vec{y})$ and k_i is the corresponding position containing x_i in the atom $S(\vec{x}, \vec{y})$. For example, the ID $\forall x\ \forall u\ R(x, u) \rightarrow \exists y\ S(y, x)$ would be written $R[1] \rightarrow S[2]$.

An *access schema* consists of:

— A collection of relations, each of a given arity. A *position* of a relation R is a number $\leq arity(R)$.
— A finite collection C of *schema constants* ("Smith", 3, ...). Informally, these represent a fixed set of values that a querier might use as test values in accesses. For example, if the user is performing a query involving the string "Smith", we would assume that "Smith" was a schema constant – but not arbitrary unrelated strings. In particular, *we will assume that all constants used in queries or constraints are schema constants*.
— For each relation R, a collection (possibly empty) of *access methods* Each method mt is associated with a collection of positions of R – the *input positions* of mt.
— integrity constraints, which we will take to be always sentences in first-order logic.

An *access* (relative to a schema as above) consists of an access method of the schema and a *binding* – a function assigning values to every input position of the method. If mt is an access method on relation R with arity n, I is an instance for a schema that includes R, and $\mathsf{AccBind}$ is a binding for mt, then the *output* or *result* of the access $(\mathsf{mt}, \mathsf{AccBind})$ on I is the set of n-tuples $\vec{t} \in I(R)$ such that $\vec{t}$ restricted to the input positions of mt is equal to $\mathsf{AccBind}$.

We will be looking for programs that interact with a datasource by generating accesses and manipulating the output with queries. As in most database settings, we will look for programs that use a restricted set of operators.

An *access command* over a schema Sch with access methods is of the form

$$T \Leftarrow_{\mathsf{OutMap}} \mathsf{mt} \Leftarrow_{\mathsf{InMap}} E$$

where: (1) E is a relational algebra expression E over some set of relations not in Sch ("temporary tables" henceforward) (2) mt is a method from Sch on some relation R (3) InMap, the *input mapping* of the command, is a function from the output attributes of E onto the input positions of mt (4) T, the *output table* of the command, is a temporary table (5) OutMap, the *output mapping* of the command, is a bijection from positions of R to attributes of T. Note that an access method may have an empty collection of inputs positions. In such a case, the corresponding access is defined over the empty binding, and an access command using the method must take the empty relation algebra expression $\emptyset$ as input. In other words, the access method makes the relation freely accessible. In Example 1.3 the $\mathsf{Udirectory}$ table was assumed to have such an "input-free" access method.

A *middleware query command* is of the form $T := Q$, where Q is a relational algebra query over temporary tables and T is a temporary table. A *return command* is of the form $\mathsf{Return}\ E$, where E is a relational algebra expression as above. An *RA-plan* consists of a sequence of access and middleware query commands, along with at most one return command.

When writing access commands we will omit the mappings for readability when they are clear from context. Returning to Example 1.3, a plan that is equivalent to the query would be represented as follows, with $\mathsf{mt}_{\mathsf{Udirectory}}$ and $\mathsf{mt}_{\mathsf{Profinfo}}$ being the methods on the corresponding tables:

$$\begin{aligned} T_1 &\Leftarrow \mathsf{mt}_{\mathsf{Udirectory}} \Leftarrow \emptyset \\ T_2 &:= \pi_1(\sigma_{\#2=\text{“}\mathsf{Smith}\text{”}} T_1) \\ T_3 &\Leftarrow \mathsf{mt}_{\mathsf{Profinfo}} \Leftarrow T_2 \\ &\mathsf{Return}\ \pi_{\mathsf{eid}} T_3 \end{aligned}$$

A temporary table is *assigned* in a plan if it occurs on the left side of a command, and otherwise is said to be *free*. The semantics of plans is defined as a function that takes as input an instance I for Sch and interpretations of the free tables. If the plan has no Return statement, the output consists of interpretations for each assigned temporary table. If the plan contains a statement $\mathsf{Return}\ E$, the output is an interpretation of a relation with attributes for each output attribute of E. In the latter case we refer to this as the *output* of the plan. An access command $T \Leftarrow_{\mathsf{OutMap}} \mathsf{mt} \Leftarrow_{\mathsf{InMap}} E$ is executed by evaluating the expression E on I and "accessing mt on every result tuple". That is, each output tuple of E is mapped to a tuple $t_{j_1} \ldots t_{j_m}$ using the input mapping InMap. For each tuple $\vec{t} = t_1 \ldots t_n \in R$ that "matches" (i.e., that extends) $t_{j_1} \ldots t_{j_m}$, $\vec{t}$ is transformed to a tuple $\vec{t'}$ using the output mapping OutMap. The interpretation of T is then the union of all such tuples $\vec{t'}$. A middleware query command $T := E$ executes query E on the contents of the temporary tables mentioned in E, and assigns the result to temporary table T.

A plan is evaluated by evaluating each command in sequence, with each command operating on the instance formed from the input instance by adding the interpretations of assigned tables produced by earlier commands. For a plan having as its final command $\mathsf{Return}\ E$ the output of the plan is the evaluation of E on the instance formed as above.

In RA-plans we allowed arbitrary relational algebra expressions in both the inputs to access commands and the middleware query commands. We can similarly talk about SPJ*-plans*, where the expressions in access and middleware query commands are built up from relational algebra operators SELECT, PROJECT, and JOIN, along with renaming and constant operators for each schema constant. $USPJ$*-plans* allow UNION in addition to SPJ operators.

We define $USPJ^{\neg}$*-plans* as any RA-plans in which relational algebra's difference operator only occurs in a *non-membership check*, which tests whether the tuples in a projection of a temporary table are not in a given relation R. Formally, a non-membership check is a sequence of two commands:

$$T' \Leftarrow_{\mathsf{OutMap}} \mathsf{mt} \Leftarrow_{\mathsf{InMap}} \pi_{a_{j_1} \ldots a_{j_m}}(T)$$
$$T'' := T - (T \bowtie T')$$

where in the first command: (1) mt is an access method on some relation R with input positions $j_1 \ldots j_m$, (2) the input mapping InMap maps attribute a_{j_i} to position j_i, (3) the attributes of the output table T' are a subset of the attributes of T and contain every a_{j_i}, (4) the output mapping OutMap maps position j_i back to a_{j_i}. In the second command, the join condition identifies attributes that have the same name.

Plans that answer queries. We now formalize the notion of a plan being "correct" for a query. Given an access schema Sch a plan *answers* a query Q *(over all instances)* if for every instance I satisfying the constraints of Sch, the output of the plan on I is the same as the output of Q. We say that the plan *answers* Q *over finite instances* if this holds for every finite instance I satisfying the constraints. *Throughout this paper we will be concerned with plans that answer a query over all instances*, and we will just say the plan answers Q (without qualification) to denote this. However, we will show that for the constraints we focus on here, there is no difference between answering over finite instances and answering over all instances.

Cost. A *plan cost function* associates every plan with a non-negative integer cost. The *minimal cost problem* for an access schema Sch and integrity constraints, query Q , and cost function Cost is the problem of finding a plan which conforms to Sch and which answers Q (with respect to constraints in Sch) while having minimal value of Cost.

The general algorithmic approach we describe can be applied with an arbitrary cost function, but our completeness results will always require strong assumptions on cost. Given a plan PL whose access commands, ordered by appearance, are $\mathsf{Command}_1 \ldots \mathsf{Command}_j$, its *method sequence* denoted $\mathsf{Methods}(\mathsf{PL})$ is the sequence of $\mathsf{mt}_1 \ldots \mathsf{mt}_j$, where mt_i is the method used in $\mathsf{Command}_i$. We say that PL *uses no more methods than* PL', denoted $\mathsf{PL} \preceq_{\mathsf{Meth}} \mathsf{PL}'$, if the method sequence of PL is a subsequence (not necessarily contiguous) of the method sequence of PL'. A cost function Cost is *simple* if $\mathsf{PL} \preceq_{\mathsf{Meth}} \mathsf{PL}'$ implies $\mathsf{Cost}(\mathsf{PL}) \leq \mathsf{Cost}(\mathsf{PL}')$. For example, a function that takes a weighted sum of the methods used in access commands within a plan would be a simple cost function.

2.1. TGDs and the chase

This work will deal with reasoning about logical formulas. A basic reasoning problem is to determine whether a φ_1 *entails* another sentence φ_2, meaning: in any instance where φ_2 holds, φ_1 holds. We can also talk about a formula $\varphi_1(\vec{x})$ entailing another formula $\varphi_2(\vec{x})$: this means that in any instance and any binding of the variables $\vec{x}$ to elements of the instance, if $\varphi_1(\vec{x})$ holds then $\varphi_2(\vec{x})$ holds. We write $\varphi_1 \models \varphi_2$ to indicate that φ_1 entails φ_2.

We recall that *proof systems* for logics are formal systems for showing that an entailment holds in the logic. A proof system is *complete* if every entailment that is true has a proof. If we use $\rho(\vec{x}) \vdash \varphi(\vec{x})$ to denote that one can prove $\varphi(\vec{x})$ from $\rho(\vec{x})$, for a complete proof system we have $\rho(\vec{x}) \models \varphi(\vec{x})$ iff $\rho(\vec{x}) \vdash \varphi(\vec{x})$.

We will be interested in special kinds of entailments, of the form

$$Q \wedge \Sigma \models Q'$$

where Q and Q' are conjunctive queries and Σ is a conjunction of TGDs.

This entailment problem is often called "query containment with constraints" in the database literature. We often say that Q is contained in Q' *w.r.t.* Σ. A specialized method has been developed for these problems, called *the chase* [Maier et al. 1979; Fagin et al. 2005].

A proof in the chase consists of a sequence of database instances, beginning with the *canonical database* of query Q: the database whose elements are the constants of Q plus copies c_1 of each variable x_1 in Q and which has a fact $R(c_1 \ldots c_n)$ for each atom $R(x_1 \ldots x_n)$ of Q. These databases evolve by *firing rules*. Given a set of facts I and a TGD $\delta = \forall x_1 \ldots x_j \; \varphi(\vec{x}) \to \exists y_1 \ldots y_k \; \rho(\vec{x}, \vec{y})$ a *trigger for* δ is a tuple $\vec{e}$ such that $\varphi(\vec{e})$ holds. An *active trigger* is one for which there is no $\vec{f}$ such that $\rho(\vec{e}, \vec{f})$ holds in I. A *rule firing* for a trigger adds facts to I that make $\rho(\vec{e}, \vec{f})$ true, where $f_1 \ldots f_k$ are new constants ("chase constants") distinct from those in the schema. Such a firing is also called a *chase step*. If the trigger was an active trigger, it is a *restricted chase step*.

A *chase sequence* following a set of dependencies Σ consists of a sequence of instances $\mathsf{config}_i : 1 \leq i \leq n$, where config_{i+1} is obtained from config_i by some rule firing of a dependency in Σ. Thus each $1 \leq i \leq n$ is associated to an instance config_i (a *chase configuration*), to a rule firing, and to a set of *generated facts* — the ones produced by the last rule firing. If Q' is a conjunctive query and config is a chase configuration having elements for each free variable of Q', then a homomorphism of Q' into config mapping each free variable into the corresponding element is called a *match* for Q' in config. A *chase proof* for the entailment $Q \wedge \Sigma \models Q'$ is a chase sequence beginning with the canonical database of Q, applying chase steps with Σ, ending in a configuration having a match.

We now have the following well-known result, saying that the chase is a complete proof system for CQ containment under constraints:

THEOREM 2.1. *[Maier et al. 1979; Fagin et al. 2005] For any instance I, for conjunctive queries Q and Q' with the same free variables, and any TGD constraints Σ, Q is contained in Q' w.r.t. Σ iff there is a chase sequence following Σ beginning with the canonical database of Q, leading to a configuration that has a match for Q'.*

Example 2.2. We recall the schema from Example 1.2, containing information about employees and departments. The constraints Σ were the following two TGDs:

$$\forall \mathsf{deptid} \; \forall \mathsf{dname} \; \forall mgrid \; \mathsf{Department}(\mathsf{deptid}, \mathsf{dname}, \mathsf{mgrid}) \to \exists N \; \mathsf{Employee}(\mathsf{mgrid}, N, \mathsf{deptid})$$
$$\forall \mathsf{eid} \; \forall \mathsf{ename} \; \forall \mathsf{deptid} \; \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid}) \to \exists D \exists M \; \mathsf{Department}(\mathsf{deptid}, D, M)$$

Consider the following two queries:

$$Q = \{\mathsf{deptid} \mid \exists \mathsf{eid}\ \exists \mathsf{ename}\ \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid})\}$$
$$Q^* = \{\mathsf{deptid} \mid \exists \mathsf{eid}\ \exists \mathsf{ename}\ \mathsf{Department}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid})\}$$

We claim that Q is contained in Q^* relative to the constraints of the schema; in the more general logical terminology, that:

$$Q \wedge \Sigma \models Q^*$$

To do this we perform a chase proof.

We begin our proof with the "canonical database" of our assumption query $Q = \exists \mathsf{eid}\ \exists \mathsf{ename}\ \mathsf{Employee}(\mathsf{eid}, \mathsf{ename}, \mathsf{deptid})$. That is, we fix constants $\mathsf{eid}_0, \mathsf{ename}_0, \mathsf{deptid}_0$ witnessing the variables to get the "initial database":

$$\mathsf{Employee}(\mathsf{eid}_0, \mathsf{ename}_0, \mathsf{deptid}_0)$$

We can now perform a "chase step" with the second integrity constraint, to derive a new fact:

$$\mathsf{Department}(\mathsf{deptid}_0, D, M)$$

where D, M are new constants.

We can now match Q^* against the set of facts we have produced, with the homomorphism mapping the free variable deptid in Q^* to the corresponding constant deptid_0.

This chase proof witnesses that Q is contained in Q^* $w.r.t.$ Σ.□

One way to find a chase proof is to "chase an initial instance as much as possible". For any set of TGDs Σ and initial instance I, we could just fire rules in an arbitrary order, making sure that any rule that is triggered fires eventually. The union of all facts generated will give an instance that satisfies the constraints, but it may be infinite. We refer to this as *the result of chasing I with* Σ. There will be many such instances depending on the order of rules fired, but they will all satisfy the same conjunctive queries by Theorem 2.1.

Sometimes one can fully chase an initial instance and get a finite chase sequence and finite final configuration. A *restricted chase sequence* is one that makes only restricted chase steps (i.e. steps using active triggers). A finite restricted chase sequence *terminates* if in the final configuration there are no active triggers. That is, eventually no rules can fire that add new witnesses. If we have a terminating chase sequence beginning with the canonical database of Q, Theorem 2.1 implies that for any conjunctive query Q' Q is contained in Q' $w.r.t.$ the constraints iff Q' has a match in the final configuration. If the constraints have the property that every long enough restricted chase sequence terminates, we will say that the constraints *have terminating chase*.

3. Expressiveness of plan languages

The language of RA-plans described in the previous section allows one to express "first-order plans" – plans that perform accesses and manipulate the results in Relational Algebra, which is known to have the expressiveness of first-order logic. We will review two other formalisms for defining plans using RA or first-order logic, and show that they give the same expressiveness as RA-plans. We will make use of this equivalence later in the paper.

Nested plans. It will sometimes be convenient to program plans with a higher-level syntax that allows a notion of subroutine. We formalize this by defining an extension of RA-plans with subroutines, the *nested RA-plans*. We inductively define the syntax of nested plans, along with the definition of a temporary table being *free* or *assigned* within a nested plan, extending the definition for RA-plans in Section 2. While for RA-plans, every temporary table mentioned in the plan will be either free or assigned, this will not be the case for nested plans.

An atomic nested plan is either: (1) an access command $T \Leftarrow_{\mathsf{OutMap}} \mathsf{mt} \Leftarrow_{\mathsf{InMap}} E$ (2) a middleware query command $T := E$ where E is an RA expression over temporary tables and $T_{\vec{x}}$. (3) a command $\mathsf{Return}\ T$, where T is a temporary table. In each case T is the only assigned temporary table of the plan, and the tables mentioned in E are free tables.

Nested plans are built up via concatenation and *subplan calls*.

If PL_1 and PL_2 are nested plans, then $\mathsf{PL}_2 \cdot \mathsf{PL}_1$ (read as "PL_2 followed by PL_1") is a nested plan. The free tables are the free tables of PL_2 along with any free tables of PL_1 that are not assigned tables of PL_2. The assigned tables of the concatenation are the assigned tables of PL_2 unioned with the assigned tables of PL_1.

If PL_1 is a nested plan which includes a Return command at "top-level" (not nested inside a subplan call), T is a free table in PL_1, E is an RA expression over temporary tables disjoint from those of PL_1 whose output matches the attributes of T, and T' is a new temporary table whose attributes are those of the output of PL_1, then

$$T' \Leftarrow \mathsf{PL}_1[T] \Leftarrow E$$

is a nested plan. The assigned tables of this plan are the assigned tables of PL_1 along with T', while the free tables are those of PL_1 minus $\{T\}$ along with any tables mentioned in E. Informally, this plan evaluates E to get a set of tuples I_E, performs PL_1 in parallel with the table T corresponding to $\{\vec{t}\}$ for each tuple $\vec{t}$ in I_E, and sets T' to be the union of each tuple $\vec{o}$ in the output of such a call.

Formally, we can define the result of an assigned temporary table T in a nested plan PL, along with the output of such a plan, when evaluated with respect to an instance I for the Sch relations and all free temporary tables of PL. The evaluation of an access command is as before, the evaluation of a middleware query command is standard, and the evaluation of $\mathsf{PL}_2 \cdot \mathsf{PL}_1$ is via evaluating PL_1 on the expansion of the input via the evaluation of tables in PL_2. The evaluation of $T' \Leftarrow \mathsf{PL}_1[T] \Leftarrow E$ is

$$\bigcup_{\vec{t} \in E(I)} \mathsf{PL}_1(I, T := \{\vec{t}\})$$

where $I, T := \{\vec{t}\}$ is the instance formed from I by interpreting T as $\{\vec{t}\}$.

Executable queries. We now introduce another approach to describing plans. In the prior literature on querying with access methods the emphasis has been on identifying syntactic restrictions on queries that guarantee that they can be implemented via access defined in an access schema. We review these notions of executable query below.

The notion of executability was first defined for conjunctive queries. A conjunctive query Q with atoms $A_1 \ldots A_n$ is *executable* relative to a schema with access patterns [Li and Chang 2000] if there is an annotation of each atom $A_i = R_i(\vec{x}_i)$ with an access method mt_i on R such that for each variable x of Q, for the first A_i containing x, x occurs only in an output position of mt_i. A UCQ $\bigvee_i Q_i$ where Q_i is a CQ is said to be executable if each disjunct is executable.

Every executable UCQ is clearly "implementable with access commands that use the given methods". In fact, every executable conjunctive query Q can be converted naïvely to an SPJ-plan $\mathsf{PlanOf}(Q)$.

PROPOSITION 3.1. *Every executable CQ can be converted to an SPJ-plan, where the number of access commands of the plan is equal to the number of atoms in the query. Similarly every executable UCQ can be converted to a $USPJ$-plan.*

PROOF. We inductively translated conjunctions of atoms to plans, with the base case translating the empty conjunction to the empty plan. The inductive rule will remove the Return command in $\mathsf{PlanOf}(A_1 \dots A_{i-1})$ and append on (1) the access command $T'_i \Leftarrow \mathsf{mt}_i \Leftarrow E_i(T_{i-1})$ where mt_i is the method annotating A_i and E_i consists of SPJ operations that project onto the input positions of mt_i and enforce repetition of variables and schema constants in input positions of mt_i, (2) the middleware command $T_i := E'_i(T'_i) \bowtie T_{i-1}$ where T_{i-1} is the output table of $\mathsf{PlanOf}(A_1 \dots A_{i-1})$, which will have attributes for all variables of $A_1 \dots A_i$, and E'_i consist of selections that enforce repetition of variables and schema constants in output positions of mt_i, (3) the command $\mathsf{Return}\ T_i$ A final projection operation will enforce any projections in Q. By translating one disjunct at a time, we see that every executable UCQ translates into a $USPJ$-plan. $\square$

We wish to extend the notion of executability to first-order queries. Although a prior definition exists in the literature [Nash and Ludäscher 2004a], we will find it useful to build our own.

An FO formula is *executable for membership checks* (relative to an access schema Sch) if it is built up from equalities and the formula True using arbitrary boolean operations and the quantifiers:

$$\forall \vec{y}\ [R(\vec{x}, \vec{y}) \rightarrow \varphi(\vec{x}, \vec{y}, \vec{z})]$$

$$\exists \vec{y}\ R(\vec{x}, \vec{y}) \wedge \varphi(\vec{x}, \vec{y}, \vec{z})$$

and for any such quantification above, if R is an Sch relation, then R has an access method mt such that all of the input positions of mt are occupied by some x_i (that is: by a variable or constant).

Notice that the definition of executable UCQ enforces restrictions related to the access methods on both quantification and the free variables. However, for formulas executable for membership checks we enforce restrictions on quantification but impose no restriction on the free variables. Thus we cannot be sure that such formulas can be implemented using the access methods. However, if we are given a tuple, we can check whether it satisfies the formula using the access methods.

Let $\varphi(\vec{x})$ be a first-order formula using the schema relations and additional tables $\vec{T}$. Let PL be an RA-plan which has output attributes for each variable in $\vec{x}$, and has free temporary tables contained in $\vec{T} \cup \{T_{\vec{x}}\}$ where $T_{\vec{x}}$ is an additional temporary table with attributes for each variable in $\vec{x}$. We say that such a plan PL *filters* φ if

$$\mathsf{PL}(I^*) = \{\vec{o}^* \mid \vec{o} \in I(T_{\vec{x}}) \wedge I, \vec{o} \models \varphi\}$$

where for a variable binding $\vec{o}$ with free variables $x_1 \dots x_n$, $\vec{o}^*$ is the corresponding tuple with attributes $a_1 \dots a_n$, and I^* is the same as I except that free tables T with arity k are considered as tuples with attributes $\#1 \dots \#k$.

PROPOSITION 3.2. *There is a linear time procedure taking as input a first-order formula φ with free variables $x_{a_1} \dots x_{a_n}$ that is executable for membership checks and producing an RA-plan with output attributes $a_1 \dots a_n$ that filters it. Furthermore, if the FO query is existential the result is a $USPJ^{\neg}$-plan while if the query is positive existential, the result is a USPJ plan.*

PROOF. We create a function $\mathsf{ToPlan}(\varphi(\vec{x}))$ that returns a plan that filters it. For simplicity we will assume that formula φ does not contain constants. The definition of ToPlan will be via induction on the structure of φ.

$\mathsf{ToPlan}(\mathsf{True})$ will be the plan that just returns $T_{\vec{x}}$ while $\mathsf{ToPlan}(x_i = x_j)$ performs a selection on $T_{\vec{x}}$. $\wedge$ and $\vee$ will translate to join and union in the usual way.

Consider the formula $\psi = \exists \vec{y}\ R(\vec{x}, \vec{y}) \wedge \varphi(\vec{x}, \vec{y}, \vec{z})$. For the active-domain semantics, it suffices to consider such "relativized quantifications", since a general existential quantification can be broken up into a union of these. Assume for simplicity that $R(\vec{x}, \vec{y})$ has no repetition of variables. $\mathsf{ToPlan}(\psi)$ will be a plan that takes as input $T_{\vec{x}\cup\vec{z}}$ with attributes corresponding to $\vec{x} \cup \vec{z}$, and consists of the concatenation of the following commands:

$$
\begin{aligned}
&T_1 \Leftarrow_{\mathsf{OutMap}} \mathsf{mt}_R \Leftarrow_{\mathsf{InMap}} \pi_{\vec{x}} T_{\vec{x}\cup\vec{z}} \\
&T_2 := T_1 \bowtie T_{\vec{x}\cup\vec{z}} \\
&T_3 := \mathsf{ToPlan}(\varphi)(T_{\vec{x}\cup\vec{y}\cup\vec{z}} := T_2) \\
&\mathsf{Return}\ \pi_{\vec{x},\vec{z}} T_3
\end{aligned}
$$

Above (1) mt_R is any access method on R such that all of its input positions are occupied by an x_i from $R(\vec{x}, \vec{y})$. Such a method exists since φ is executable for membership checks. (2) $T_3 := \mathsf{ToPlan}(\varphi)(T_{\vec{x}\cup\vec{y}\cup\vec{z}} := T_2)$ is the set of commands in $\mathsf{ToPlan}(\varphi)$ with the table T_2 substituted for $T_{\vec{x}\cup\vec{y}\cup\vec{z}}$ and an assignment to T_3 replacing the Return command. (3) InMap maps attribute a_{x_i} of $T_{\vec{x}\cup\vec{z}}$ to the position of R containing x_i. (4) OutMap maps position i of R to attribute x_{a_j} or y_{a_j}, where x_{a_j} or y_{a_j} is in position i of R in $R(\vec{x}, \vec{y})$. The case where variables are repeated is handled by inserting additional middleware query commands that enforce these repetitions.

To compute $\mathsf{ToPlan}(\forall \vec{y}\ R(\vec{x}, \vec{y}) \rightarrow \varphi(\vec{x}, \vec{y}, \vec{z}))$, it suffices to get a plan for its negation $\exists \vec{y}\ R(\vec{x}, \vec{y}) \wedge \neg\varphi(\vec{x}, \vec{y}, \vec{z}))$. Thus we give a construction for the case of general negation.

$\mathsf{ToPlan}(\neg\varphi)$ returns $T_{\vec{x}} - \mathsf{ToPlan}(\varphi)$, where $T_{\vec{x}}$ has attributes corresponding to the free variables of φ. This can be implemented by a plan that first performs the commands in $\mathsf{ToPlan}(\varphi)$, with the output in some table T', and then does a middleware query command subtracting T' from $T_{\vec{x}}$. When φ is a relational atom, this can be implemented as a non-membership check.

The properties of the translation are easily verified. □

An *executable FO query* will be a query that performs an executable UCQ to get a set of tuples, and then filters it using a formula executable for membership checks. Formally such a query consists of:

(i) a set $x_1 \dots x_k$ of variables
(i) a first order formula $\tau(x_1 \dots x_l)$ using a distinguished relation $T_{\vec{x}}$, containing as free variables each $x_1 \dots x_k$, whose arity matches the number of free variables in τ, with τ executable for membership checks;
(i) an executable UCQ $\epsilon(x_1 \dots x_l)$;

To evaluate an executable FO query on an instance I, we proceed as follows: (1) evaluate ϵ over I to get a set of tuples $I_{\vec{x}}$ (2) evaluate τ over the instance formed from I by making $I_{\vec{x}}$ the interpretation of $T_{\vec{x}}$ to get a subset $I'_{\vec{x}}$ of the tuples in $I_{\vec{x}}$ (3) project $I'_{\vec{x}}$ on $x_1 \dots x_k$ We refer to $x_1 \dots x_k$ as the *return variables*, ϵ as the *output envelope* and τ as the *filter formula*.

Expressive equivalence. We now compare the languages we have introduced.

From Propositions 3.1 and 3.2 we see that

PROPOSITION 3.3. *Every executable FO query can be converted into an RA-plan.*

We will now explain that, conversely, nested RA-plans can be translated into executable FO queries, and thus the same is true for RA-plans. This will imply that RA-plans, nested RA-plans, and executable FO queries have the same expressiveness.

In the electronic appendix, we show:

THEOREM 3.4. *Nested RA plans, RA plans, and executable FO queries have the same expressiveness, and there are computable transformations going from each formalism to an equivalent query in the other.*

We also show in the electronic appendix that every $USPJ^{\neg}$-plan has an equivalent $USPJ^{\neg}$-query, and hence (by prior results) an equivalent existential formula. We also show, using results of [Deutsch et al. 2007], that the notion captures all existential formulas that have a plan.

4. Axiomatizing access patterns

We will now present "proof goals that capture the existence of a plan". That is, we will give a set of axioms and a proof goal using those axioms such that proofs that realize the proof goal can be converted to plans. We will need different proof goals for different kinds of plans. We will start with a proof goal that will correspond to plans that do not use negation, the SPJ-plans.

Given schema Sch, the *forward accessible schema for* Sch, denoted $\mathsf{AcSch}(\mathsf{Sch})$, is the schema without any access restrictions, such that:

— The constants are those of Sch.
— The relations are those of Sch, a unary relation $\mathsf{accessible}(x)$ (x is an *accessible value*) plus a copy of each relation R of Sch called $\mathsf{InfAcc}R$ (the *inferred accessible* version of R).
— The constraints are those of Sch (referred to as "Sch constraints" below) along with the following constraints
  — *accessibility axioms*: for each access method mt on relation R of arity n with input positions $j_1 \ldots j_m$ we have a rule:

$$\mathsf{accessible}(x_{j_1}) \wedge \ldots \wedge \mathsf{accessible}(x_{j_m}) \wedge R(x_1 \ldots x_n) \rightarrow \mathsf{InfAcc}R(x_1 \ldots x_n) \wedge \bigwedge_j \mathsf{accessible}(x_j)$$

  In addition, we have $\mathsf{accessible}(c)$ for each constant c of Sch.
  — A copy of each of the original integrity constraints, with each relation R replaced by $\mathsf{InfAcc}R$. We refer to these as "$\mathsf{InfAccCopy}$ constraints" below.

Informally, $\mathsf{accessible}(c)$ indicates that the value c can be returned by some sequence of accesses. The inferred accessible relations represent facts that can be derived from facts exposed via the access methods using reasoning. Thus the forward accessible schema represents the rules that allow one to move from a "hidden fact" (e.g. $R(c_1 \ldots c_n)$) to an inferred accessible fact (e.g. $\mathsf{InfAcc}R(c_1 \ldots c_n)$), and from there — using the constraints — to other inferred accessible facts (e.g. $\mathsf{InfAcc}S(c_1 \ldots c_n, d)$ for a new chase constant d, witnessing the right-hand side of a rule firing requiring $\exists y\ \mathsf{InfAcc}S(c_1 \ldots c_n, y)$. From the structure of the rules one sees that an $\mathsf{InfAccCopy}$ constraint can fire based upon facts generated by other kinds of rules, but the firing of an $\mathsf{InfAccCopy}$ constraint can not trigger either accessibility axioms or Sch constraints.

Given a query Q, its *inferred accessible version* $\mathsf{InfAcc}Q$ is obtained by replacing each relation R by $\mathsf{InfAcc}R$. Informally, $\mathsf{InfAcc}Q$ represents the fact that the existence of a witness to Q can be obtained through making accesses and reasoning.

We will overload $\mathsf{AcSch}(\mathsf{Sch})$ to refer to the conjunction of axioms in this schema $\mathsf{AcSch}(\mathsf{Sch})$. For a positive existential plan we will be interested in the entailment:

$$Q \wedge \mathsf{AcSch}(\mathsf{Sch}) \models \mathsf{InfAcc}Q$$

Informally, this means that we can infer from Q holding in a hidden database that Q's truth must be visible to a user via accesses and reasoning with constraints.

Example 4.1. Recall the setting of Example 1.3. We had a $\mathsf{Profinfo}$ table containing information about faculty, including their last name, office number, and their id, with a restricted interface that requires giving an id of an employee as an input. We also had a $\mathsf{Udirectory}$ table containing the id and last name of every university employee, with an input-free access method.

We were interested in the query asking for ids of faculty named "Smith". That is:

$$Q = \exists \mathsf{onum}\ \mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \text{“Smith”})$$

In this case we have:

$$\mathsf{InfAcc}Q = \exists\ \mathsf{onum}\ \mathsf{InfAccProfinfo}(\mathsf{eid}, \mathsf{onum}, \text{“Smith”}).$$

The forward accessible schema includes rules

- $\mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \mathsf{lname}) \rightarrow \mathsf{Udirectory}(\mathsf{eid}, \mathsf{lname})$
- $\mathsf{Udirectory}(\mathsf{eid}, \mathsf{lname}) \rightarrow \mathsf{InfAccUdirectory}(\mathsf{eid}, \mathsf{lname}) \wedge \mathsf{accessible}(\mathsf{lname}) \wedge \mathsf{accessible}(\mathsf{eid})$
- $\mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \mathsf{lname}) \wedge \mathsf{accessible}(\mathsf{eid}) \rightarrow \mathsf{InfAccProfinfo}(\mathsf{eid}, \mathsf{onum}, \mathsf{lname}) \wedge \mathsf{accessible}(\mathsf{onum}) \wedge \mathsf{accessible}(\mathsf{lname})$

One can check that Q is contained in $\mathsf{InfAcc}Q$ *w.r.t.* $\mathsf{AcSch}(\mathsf{Sch})$. □

In $\mathsf{AcSch}(\mathsf{Sch})$ we only had rules going from the original relations R to $\mathsf{InfAcc}R$. Going back to the informal intuition, we can think of this as capturing the positive information revealed in an access, but not the negative information (that a certain tuple is *not* in the answer). We will later prove that this entailment is equivalent to existence of an SPJ-plan. To capture first-order plans, we should have axioms capturing both positive and negative information returned by accesses. We give such an extension now.

Let $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ extend the axioms of $\mathsf{AcSch}(\mathsf{Sch})$ with the following axioms (universal quantifiers omitted):

$$\bigwedge_{i \le m} \mathsf{accessible}(x_{j_i}) \wedge \mathsf{InfAcc}R(x_1 \dots x_n) \rightarrow R(x_1 \dots x_n) \wedge \bigwedge_{i \le n} \mathsf{accessible}(x_i)$$

Above, R is a relation of Sch having an access method with input positions $j_1 \dots j_m$. Notice that these rules are obtained from those of $\mathsf{AcSch}(\mathsf{Sch})$ by switching the roles of $\mathsf{InfAcc}R$ and R, resulting in a rule set where the original schema and the InfAcc copy are treated symmetrically. We will now be interested in the entailment:

$$Q \wedge \mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch}) \models \mathsf{InfAcc}Q$$

We can think informally of $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ as capturing both positive and negative information revealed from an access.

4.1. Statement of the main results

We are now ready to state our main results on the relationship between the entailments mentioned before and plans. We will also explain how each entailment is stating a semantic property of the query Q.

RA-plans and the schema $\mathsf{AcSch}^{\leftrightarrow}$. For RA-plans, our main result is:

THEOREM 4.2. *For any conjunctive query Q and access schema Sch with TGD constraints, there is an RA-plan answering Q (over databases in Sch) if and only if $Q \wedge \mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch}) \models \mathsf{InfAcc}Q$.*

Further, from any chase proof witnessing $Q \wedge \mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch}) \models \mathsf{InfAcc}Q$ we can extract (in linear time) an RA-plan for Q over Sch.

We now wish to explain the semantic property, which we will denote as *access-determinacy*, that corresponds to the proof goal $Q \wedge \mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch}) \models \mathsf{InfAcc}Q$, and (as we will later show) to the existence of an RA-plan.

Given an instance I for schema Sch the *accessible part of* I, denote $\mathsf{AccPart}(I)$ consists of all the facts over I that can be obtained by starting with empty relations and iteratively entering values into the access methods. Formally, it is a database containing a set of facts $\mathsf{Accessed}R(v_1 \ldots v_n)$, where R is a relation and $v_1 \ldots v_n$ are values in the domain of I such that $R(v_1 \ldots v_n)$ holds in I, obtained by starting with relations $\mathsf{Accessed}R_0$ and $\mathsf{accessible}_0$ empty[1], and then iterating the following process until a fixpoint is reached:

$$\mathsf{accessible}_{i+1} = \mathsf{accessible}_i \;\cup \bigcup_{\substack{R \text{ a relation} \\ j<arity(R)}} \pi_j(\mathsf{Accessed}R_i)$$

and

$$\mathsf{Accessed}R_{i+1} = \mathsf{Accessed}R_i \;\cup \bigcup_{\substack{(R,\{j_1,\ldots,j_m\}) \\ \text{there is a method of } \mathsf{Sch} \text{ on } R \text{ with inputs } j_1,\ldots,j_m}} \{(v_1 \ldots v_n) \in I(R) |\; v_{j_1} \ldots v_{j_m} \in \mathsf{accessible}_i\}$$

Above $\pi_j(\mathsf{Accessed}_i(R))$ denotes projection of $\mathsf{Accessed}_i(R)$ on the j^{th} position. For finite instances a fixpoint will be reached after $|I|$ steps, where $|I|$ denotes the number of facts in I. For arbitrary instances the limit of these instances over all i will be a fixpoint.

Above we consider $\mathsf{AccPart}(I)$ as a database instance for the schema with relations $\mathsf{accessible}$ and $\mathsf{Accessed}R$. Below we will sometimes refer to the values in the relation $\mathsf{accessible}$ as the *accessible values of* I.

In the case of vocabulary-based access-restrictions, the accessible part of an instance just represents the restriction of the instance to the visible relations (e.g. view tables). We now are ready to give the semantic property.

Definition 4.3 (*Access-determinacy*). Q is said to be *access-determined* over Sch if for all instances I and I' satisfying the constraints of Sch with $\mathsf{AccPart}(I) = \mathsf{AccPart}(I')$ we have $Q(I) = Q(I')$.

If a query is *not* access-determined, it is obvious that it cannot be answered through any plan, since it is easy to see that any plan can only read tuples from the accessible part. In the case of interface restrictions given by a collection of views, each associated with a view definition, access-determinacy just says that for instance of the schema where the views are evaluated according to their definitions, the query result is a function of the view images. This is the notion of *determinacy* whose study was initiated by Segoufin and Vianu [Segoufin and Vianu 2005] and by Nash, Segoufin, and Vianu [Nash et al. 2010].

The following claim relates our entailment hypothesis and this preservation property.

CLAIM 1. *The following are equivalent, for any first order query Q and any access schema with first-order constraints:*

(*1*) Q *entails* $\mathsf{InfAcc}Q$ *with respect to the rules in* $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$

[1]In the presence of schema constants, we would start with $\mathsf{accessible}_0$ consisting of the schema constants.

(2) Q *is access-determined over* Sch

PROOF. For simplicity we assume Q is a boolean conjunctive query. The non-boolean case is a straightforward generalization.

We prove that the first item implies the second. Fix I and I' satisfying the schema with the same accessible part, and assume I satisfies Q. Consider the instance I'' for $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ formed by interpreting the relations R as in I, the relation accessible by the accessible values of I, and each $\mathsf{InfAcc}R$ by the interpretation of R in I'. Then one can easily verify that I'' satisfies the constraints of $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$.

Since I (and hence I'') satisfies Q, and we are assuming that Q entails $\mathsf{InfAcc}Q$ with respect to $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ we can conclude that I'' must satisfy $\mathsf{InfAcc}Q$. Thus Q holds in I' as required.

We now argue from the second item to the first, which will complete the proof of the claim. Suppose Q is not contained in $\mathsf{InfAcc}Q$ with respect to the rules in $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$. Hence there is an instance $I^{\mathsf{AcSch}^{\leftrightarrow}}$ satisfying the rules of $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ and also satisfying $Q \wedge \neg\mathsf{InfAcc}Q$. Let I_1 consist of the restriction of $I^{\mathsf{AcSch}^{\leftrightarrow}}$ to the original schema relations. Let I_2 consist of the inferred accessible relations from $I^{\mathsf{AcSch}^{\leftrightarrow}}$, renamed to the original schema. We first claim that a fact $F = R(e_1 \ldots e_n)$ of the accessible part of I_1 is in the accessible part of I_2. We prove this by induction on the *appearance point of* F, the lowest i such that F appears in $\mathsf{Accessed}R_i$. Since $\mathsf{Accessed}R$ in the accessible part is the union of these relations, a minimal i must exist for each F. F appears in $\mathsf{Accessed}R_i$ due to an access using elements $e_{j_1} \ldots e_{j_m}$ that satisfy accessible facts that had a strictly smaller appearance point. Thus by induction these earlier facts are in the accessible part of I_2, and in particular $e_{j_1} \ldots e_{j_m}$ are accessible values of I_2. Using the axioms we have that $\mathsf{InfAcc}R(e_1 \ldots e_n)$ holds, and thus $R(e_1 \ldots e_n)$ holds in I_2. Using the definition of accessible part, we conclude that F is in the accessible part of I_2 as required. Arguing symmetrically, we have that I_1 and I_2 have the same accessible part, and hence they contradict access-determinacy. □

Using the above claim, we can restate Theorem 4.2:

For any conjunctive query Q and access schema Sch with TGD constraints there is an RA-plan answering Q (over databases in Sch) if and only if Q entails $\mathsf{InfAcc}Q$ with respect to the rules in $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ if and only if Q is access-determined.

Note that in the direction from right to left we are moving from a preservation property to a syntactic restriction.

We will return to the algorithm that proves Theorem 4.2 later.

SPJ-plans and the schema AcSch. We now state an analogous result for SPJ-plans.

THEOREM 4.4. *For any conjunctive query Q and access schema* Sch *with TGD constraints, there is an SPJ-plan answering Q (over instances in* Sch*) if and only if* $Q \wedge \mathsf{AcSch}(\mathsf{Sch}) \models \mathsf{InfAcc}Q$.

Further, for every chase proof witnessing $Q \wedge \mathsf{AcSch}(\mathsf{Sch}) \models \mathsf{InfAcc}Q$*, we can extract a SPJ-plan.*

We will again translate the entailment into a preservation property of the query Q.

Definition 4.5 (*Access monotonic determinacy*). We say Q is *access-monotonically-determined* over Sch if for all instances I and I' satisfying the constraints of Sch with every fact of $\mathsf{AccPart}(I)$ contained in $\mathsf{AccPart}(I')$ (that is, $\mathsf{AccPart}(I)$ is a subinstance of $\mathsf{AccPart}(I')$), then $Q(I) \subseteq Q(I')$.

That is, we have weakened the hypothesis of access-determinacy to require only containment of facts, not equality. This definition also generalizes one studied by Nash, Segoufin, and Vianu [Nash et al. 2010] in the context of constraints associated to view definitions, denoted there as *monotonicity*.

The following claim now relates these notions to our axioms, analogously to Claim 1.

CLAIM 2. *The following are equivalent (for any first order query Q and access schema with first-order constraints):*

(*1*) Q *entails* $\mathsf{InfAcc}Q$ *with respect to the constraints in* $\mathsf{AcSch}(\mathsf{Sch})$
(*2*) Q *is access-monotonically-determined w.r.t.* Sch

PROOF. Again, we assume Q is boolean for simplicity.

We prove that the first item implies the second, using the same template as in the proof of Claim 1. Fix I and I' satisfying the schema with the same accessible part, and assume I satisfies Q. Consider the instance I'' for the accessible schema formed by interpreting the relations R as in I, $\mathsf{accessible}$ by the accessible values of I, and each $\mathsf{InfAcc}R$ by the interpretation of R in I'. Access-monotonicity implies that I'' satisfies the constraints of $\mathsf{AcSch}(\mathsf{Sch})$. Since I (and hence I'') satisfies Q, the assumption tells us that I'' must satisfy $\mathsf{InfAcc}Q$, and thus Q holds in I' as required.

Arguing from the second item to the first is also analogous. Suppose Q does not imply $\mathsf{InfAcc}Q$ with respect to the rules in $\mathsf{AcSch}(\mathsf{Sch})$. Hence there is an instance I^{AcSch} satisfying the rules of $\mathsf{AcSch}(\mathsf{Sch})$ and also satisfying $Q \wedge \neg\mathsf{InfAcc}Q$. Let I_1 consist of the restriction of I^{AcSch} to the original schema relations. Let I_2 consist of the inferred accessible relations from I^{AcSch}, renamed to the original schema. We claim that a fact $R(e_1 \dots e_n)$ of the accessible part of I_1 is again in the accessible part of I_2. This proof is as in Claim 1, since in this part of the argument we only used the "forward accessibility axioms". From this, we can see that I_1 and I_2 witness that Q is not access-monotonically-determined, which completes the argument. $\square$

Thus Theorem 4.4 can be restated as:

For any conjunctive query Q and access schema Sch with TGD constraints, there is a SPJ-plan answering Q (over instances in Sch) if and only if Q entails $\mathsf{InfAcc}Q$ with respect to $\mathsf{AcSch}(\mathsf{Sch})$ iff Q is access-monotonically-determined.

$USPJ^{\neg}$**-plans.** We now investigate the situation for $USPJ^{\neg}$-plans. For arbitrary first-order constraints, there can be conjunctive queries that have $USPJ^{\neg}$-plans, but which do not have plans without use of negation. For example, consider query (x), constraints asserting $R(x) \leftrightarrow A(x) \wedge \neg B(x)$, and assume we have input-free access methods on $\{A, B\}$. Then there is a $USPJ^{\neg}$-plan that is equivalent to $A(x) \wedge \neg B(x)$, but there is no SPJ-plan.

In the conference version [Benedikt et al. 2014] we give another variant of the accessible schema that is geared towards $USPJ^{\neg}$-plans. This variant restricts the "backward accessibility axiom" so that it only applies to facts $\mathsf{InfAcc}R(\vec{d})$ with each $d_1 \dots d_n$ all satisfying $\mathsf{accessible}$. Since this paper does not deal with general first-order constraints, we do not give a proof of this result. Instead, we will focus on the TGD case, and will show that for these constraints whenever we can obtain a $USPJ^{\neg}$-plan, we can actually get an SPJ-plan as well. That is we prove, assuming Theorem 4.4,

THEOREM 4.6. *For any conjunctive query Q and access schema* Sch *with TGD constraints, if there is a $USPJ^{\neg}$-plan answering Q w.r.t.* Sch*, then there is an SPJ-plan answering Q w.r.t.* Sch.

Clearly the second item implies the first. Our proof that the second implies the first will make use of the following result:

LEMMA 4.7. *For CQ Q and access schema with TGD constraints* Sch*, if there is $USPJ^{\neg}$-plan answering Q, then there is an $USPJ$-plan answering Q.*

Assuming this lemma, Theorem 4.6 will follow from Theorem 4.4, since $USPJ$-plans are access-monotonically-determined (since all of their operations are monotone) and Theorem 4.4 states that access-monotonically-determined queries have SPJ-plans.

PROOF OF LEMMA 4.7. Let PL be a $USPJ^{\neg}$-plan for Q. Let PL$'$ be obtained from PL simply by *dropping all non-membership tests*.That is, eliminating any non-membership check

$$T' \Leftarrow \mathsf{mt} \Leftarrow \pi_{a_{j_1}\dots a_{j_m}}(T)$$
$$T'' := T - (T \bowtie T')$$

and replacing any reference to T'' with a reference to T.

A straightforward induction shows that, on all instances I, we have that PL$'$ returns (at least) all tuples returned by PL. Therefore, it suffices to show that, on all instances I satisfying Σ, every tuple $\mathbf{a}$ returned by PL$'$ is returned also by PL.

We may assume that all occurrences of the union operator in PL are at the top-most level, i.e., the only occurrence of union is in a final query middleware command that creates the output table of PL, and the tables $T_1 \dots T_n$ unioned are formed by disjoint subplans PL_i of PL. This is argued by the same technique of pulling unions to top level in $USPJ$ queries. The plans $\mathsf{PL}_1 \dots \mathsf{PL}_n$ will be called *components* of PL. Likewise let $\mathsf{PL}'_1, \dots, \mathsf{PL}'_n$ be the components of PL$'$. We see that each PL'_i is obtained from PL_i by dropping all non-membership checks. PL$'$ is a $USPJ$-plan that returns every tuple of PL. Thus to prove the lemma it suffices to show that for any I satisfying the constraints, the output of each PL'_i on I is contained in the output of PL_i. Recalling that PL was assumed to answer Q, it is sufficient to show that the output of PL_i is contained in that of Q. Let Q'_i be the conjunctive query associated with PL'_i. Since Q'_i and Q are both CQs, the completeness of the chase, Theorem 2.1, tells us that to show containment of Q'_i in Q we must show that in the instance $\widehat{I'_i}$ formed by chasing the canonical database I'_i of Q'_i with the constraints Σ, Q holds on the tuple $\mathbf{x}$ corresponding to the free variables of Q'_i in I'_i.

Let Q_i be the existential query associated with PL_i (including all negated atoms). Suppose for any of the negated atoms $\neg A_j$ of Q'_i, $A_j(\mathbf{x})$ held in $\widehat{I'_i}$. Then by Theorem 2.1 again, Q'_i entails that atom relative to the constraints. But then PL_i would be equivalent to false $w.r.t.$ Σ, and could have been removed from the list of components of PL. Thus we can assume without loss of generality that the existential query Q'_i holds of $\mathbf{x}$ in the chase instance $\widehat{I'_i}$, as required.

Thus we have shown that Q has a $USPJ$-plan. □

4.2. Summary

We have now presented a semantic property (access-monotonic-determinacy) and an entailment (entailment of InfAcc$Q$ from Q in the schema AcSch(Sch)) and we have shown they are equivalent. We have also stated a theorem that for TGDs, both of these properties are equivalent to the existence of an SPJ-plan.

Similarly, we have presented the semantic property of access-determinacy along with an entailment with respect to $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$, and shown they are equivalent, again for arbitrary constraints and queries. We have stated a theorem that these are equivalent to the existent of an RA-plan.

Finally, we have shown, assuming the characterization theorem stated for SPJ-plans above, that the existence of a $USPJ^{\neg}$-plan is equivalent to the existence of an SPJ-plan.

The proofs of the main theorems stated but not proven in this section, Theorem 4.2 and Theorem 4.4, will be given in the next section of the paper. While the claims relating entailments and semantic properties in this section held for general first-order constraints, the next section will be focused on TGDs, and will use methods specific to them.

Alternative Axiomatization of $\mathsf{AcSch}^{\leftrightarrow}$**.** Before closing this section, we present a slight alteration of the axiom schema $\mathsf{AcSch}^{\leftrightarrow}$, denoted $\mathsf{AltAccSch}^{\leftrightarrow}$ in which the predicate accessible does not appear. This axiomatization will be useful later on, and also gives insight into what the predicate accessible "really" means.

In every forward accessibility axiom an atom $\mathsf{accessible}(x)$ on the left is replaced (in all possible ways) by a relation $\mathsf{InfAcc}R(\vec{z})$, where $\vec{z}$ contains x in at least one position (and the other variables are universally quantified), while the occurrences on the right are dropped. For example, the axiom $\mathsf{accessible}(x) \wedge R(x, y) \rightarrow \mathsf{InfAcc}R(x, y) \wedge \mathsf{accessible}(y)$ would be replaced by many axioms, including $\mathsf{InfAcc}S(x, w, z) \wedge R(x, y) \rightarrow \mathsf{InfAcc}R(x, y)$.

In every backward accessibility axiom, we similarly replace $\mathsf{accessible}(x)$ on the left by an atom in the original schema containing x, while dropping occurrences on the right.

PROPOSITION 4.8. *Q proves $\mathsf{InfAcc}Q$ using the axioms of $\mathsf{AcSch}^{\leftrightarrow}$ iff Q proves $\mathsf{InfAcc}(Q)$ in the modified schema $\mathsf{AltAccSch}^{\leftrightarrow}$.*

PROOF. In one direction, suppose that Q does not prove $\mathsf{InfAcc}Q$ using the axioms of $\mathsf{AcSch}^{\leftrightarrow}$. Then by Claim 1, Q is not access-determined. Fix I and I' instances for the original schema satisfying Σ with the same accessible part, but disagreeing on the output of Q. Fix $\vec{d}$ that is returned by Q in I but not returned by Q in I'. By applying an isomorphism to non-accessible values of I' and I, we can assume that every non-accessible value of I' is not in I and vice versa. Let I^* be the instance for the augmented schema in which relations R are interpreted as in I and relations $\mathsf{InfAcc}R$ are interpreted as in I'.

We claim that I^* satisfies $\mathsf{AltAccSch}^{\leftrightarrow}$. Clearly both the original relations and the relations $\mathsf{InfAcc}R$ satisfy Σ. Consider a modified forward axiom (universal quantifiers omitted):

$$\mathsf{InfAcc}R_{j_1}(\ldots x_{j_1} \ldots) \wedge \ldots \mathsf{InfAcc}R_{j_m}(\ldots x_{j_m} \ldots) \wedge R(\vec{x}) \rightarrow \mathsf{InfAcc}R(\vec{x})$$

where R has an access method on positions $j_1 \ldots j_m$. Suppose we have a tuple $c_1 \ldots c_m$ in I^* satisfying the left-hand side of this implication. Then $c_{j_1} \ldots c_{j_m}$ must be accessible values of I and I'. Since the fact $R(\vec{c})$ holds in I, it must be in the accessible part of I, and hence in the accessible part of I'. Thus $\mathsf{InfAcc}R(\vec{c})$ holds in I^* as required. The backward accessibility axioms are argued symmetrically.

Clearly, the formula $Q(\vec{d})$ is true on I^*, since it is true in I and Q is a formula using the relations in the original schema. $\vec{d}$ does not satisfy Q in I', thus does not satisfy the formula obtained by interpreting each relation R by $\mathsf{InfAcc}R$ in I'. Thus it can not be returned by $\mathsf{InfAcc}Q$ on I^*.

I^* therefore is a witness that Q does not prove $\mathsf{InfAcc}Q$ in $\mathsf{AltAccSch}^{\leftrightarrow}$.

In the other direction, suppose we have I^* witnessing that Q does not prove $\mathsf{InfAcc}(Q)$ in $\mathsf{AltAccSch}^{\leftrightarrow}$. Expand I^* to an instance I^+ for the signature extended with accessible, by interpreting accessible by all values that lie in the domain of some $\mathsf{InfAcc}R$ relation and in the domain of some relation of the original schema. We claim that the resulting instance satisfies the constraints of $\mathsf{AcSch}^{\leftrightarrow}$. The accessibility axioms follow directly from the corresponding axioms of $\mathsf{AltAccSch}^{\leftrightarrow}$. Similarly, we see that the output of Q in I^+ is the same as the output of Q in I, while the output of $\mathsf{InfAcc}Q$ on I^+ is the same as the output of $\mathsf{InfAcc}Q$ on I^*. Thus we see that Q does not prove $\mathsf{InfAcc}Q$ in $\mathsf{AcSch}^{\leftrightarrow}$. $\square$

5. Algorithms transforming proofs to plans

Recall from Subsection 2.1 that for query containment problems using conjunctive queries and TGDs, we can use the proof system known as the chase. In the chase, a proof can be rephrased as a sequence of database instances, beginning with the canonical database of query Q, evolving by firing rules – that is, grounding the TGDs. By a *full proof* we mean a chase sequence beginning with the canonical database of the source query Q and ends with a configuration having a match for the target query.

We also have seen a collection of proof goals that capture semantic properties of queries, with respect to TGD constraints Σ. These proof goals are all of the form.

$$Q \wedge \Gamma \models \mathsf{InfAcc}Q$$

where Q is the query we are trying to reformulate, $\mathsf{InfAcc}Q$ is a query obtained from copying Q on the "inferred accessible" relations and Γ consists of two copies of Σ — the Sch constraints and $\mathsf{InfAccCopy}$ constraints — along with accessibility axioms relating the two copies. In particular, we have a problem of conjunctive query containment with respect to TGD constraints (namely Γ), and hence in seeing whether these properties hold, we can restrict our attention to chase proofs.

We will show that these semantic properties are equivalent to the existence of plans. In each case, we show that from any proof of a proof goal, we can read off a plan. We focus first on the case of SPJ-plans and the forward accessible schema $\mathsf{AcSch}(\mathsf{Sch})$.

Given a chase sequence $\mathsf{config}_1 \ldots \mathsf{config}_k$ corresponding to a full proof, let $C_{\mathsf{SeqConsts}}$ be the set of chase constants generated by firings of Sch constraints within this sequence.

We will convert prefixes of our full proof $\mathsf{config}_1 \ldots \mathsf{config}_k$ into plans by induction on the number of accessibility axioms fired in the prefix. We will generate plans PL_i for any prefix $\mathsf{config}_1 \ldots \mathsf{config}_k$ that ends with the firing of an accessibility axiom, where The generated plan PL_i will assign tuples to a temporary table T_j whose attributes correspond to a subset C_i of $C_{\mathsf{SeqConsts}}$. Informally, rows of these tables will store possible homomorphisms that map the chase constants into the instance being queried. Therefore, we interchangeably talk about constants or attributes when referring to the elements of C_i hereafter. The C_i will be monotonic under inclusion as i increases. We will maintain as an invariant that the attributes in C_i are exactly the constants $c \in C_{\mathsf{SeqConsts}}$ such that $\mathsf{accessible}(c)$ holds in the configuration of the last element of the sequence.

Our final plan will correspond to the trivial prefix $\mathsf{config}_1 \ldots \mathsf{config}_k$.

The induction step for rule firings other than accessibility axioms will append nothing to the plan. In the induction step, we consider a chase sequence ending with the firing of an accessibility axiom:

$$\begin{aligned} &\mathsf{accessible}(c_{j_1}) \wedge \ldots \mathsf{accessible}(c_{j_m}) \wedge R(c_1 \ldots c_n) \\ &\qquad \rightarrow \mathsf{InfAcc}R(c_1 \ldots c_n) \wedge \bigwedge_j \mathsf{accessible}(c_j) \end{aligned}$$

associated with method mt on relation R having input positions $j_1 \ldots j_m$. We say that the rule firing *exposes fact* $R(c_1 \ldots c_n)$. Let config_{i-1} be the chase configuration prior to the firing of this rule. Note that by the inductive invariant, each $c_{j_1} \ldots c_{j_m}$ must be an attribute of table T_{i-1} associated to the sequence prior to the firing. We will define the *commands that correspond to this rule firing*.

We will explain the induction step first in the case where none of $c_{j_1} \ldots c_{j_m}$ are schema constants and no constant is repeated in $R(c_1 \ldots c_n)$. We defer the additional cases until the next subsection. We first generate an access command whose input expression is the projection of T_{i-1} onto $c_{j_1} \ldots c_{j_m}$, with the input mapping InMap taking columns $c_{j_1} \ldots c_{j_m}$ of T_{i-1} to input positions $j_1 \ldots j_m$ of mt. The command's output will be a table T'_i with attributes $C_i = C_{i-1} \cup \{c_1 \ldots c_n\}$, with the output mapping taking position d to c_d. We follow the access command by a middleware query command that sets T_i to the join of T'_i with T_{i-1}.

If we have a full chase proof, the final configuration must have a match for $\mathsf{InfAcc}Q$. Let V be the set of chase constants corresponding to the free variables of Q. We add a last Return command that will return the projection of T_i on V. In the special case that Q is boolean, the final query amounts to checking that the table T_i is non-empty.

Algorithm 1: Chase-Proof-to-SPJ-Plan algorithm

Input: full chase proof with configurations $\mathsf{config}_1 \ldots \mathsf{config}_k$
V = attributes for free variables of Q;
$\mathsf{Plan} := \emptyset$;
T_0 = table with no columns ;
$numsteps = 0$;
for i = 1 *to* k **do**
  if config_i *is obtained by firing an accessible axiom exposing fact* $F = R(c_1 \ldots c_n)$ *via method* mt *with inputs* $j_1 \ldots j_m$ **then**
    Append to Plan command $T'_i \Leftarrow \mathsf{mt} \Leftarrow \pi_{c_{j_1} \ldots c_{j_m}}(T_{i-1})$;
    Append to Plan command $T_i = T'_i \bowtie T_{i-1}$;
    $numsteps++$
Add command "Return $\pi_V(T_{numsteps})$" to Plan ;
Return Plan

Example 5.1. Consider a variant of Example 1.3, using the same schema and the query $Q = \exists \mathsf{eid}\ \exists \mathsf{onum}\ \exists \mathsf{lname}\ \mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \mathsf{lname})$. Using the chase, we get the following **proof**:

(1) Create the canonical database, containing the single fact $\{\mathsf{Profinfo}(\mathsf{eid}_0, \mathsf{onum}_0, \mathsf{lname}_0)\}$
(2) One of the initial integrity constraints matches $\mathsf{Profinfo}(\mathsf{eid}_0, \mathsf{onum}_0, \mathsf{lname}_0)$, and firing the rule infers $\mathsf{Udirect}(\mathsf{eid}_0, \mathsf{lname}_0)$.
(3) $\mathsf{Udirect}(\mathsf{eid}_0, \mathsf{lname}_0)$ matches an accessibility axiom, and the rule firing generates $\mathsf{InfAccUdirect}(\mathsf{eid}_0, \mathsf{lname}_0)$ and $\mathsf{accessible}(eid_0)$.
(4) An accessibility axiom matches $\mathsf{Profinfo}(\mathsf{eid}_0, \mathsf{onum}_0, \mathsf{lname}_0) \wedge \mathsf{accessible}(\mathsf{eid}_0)$ creating fact $\mathsf{InfAccProfinfo}(\mathsf{eid}_0, \mathsf{onum}_0, \mathsf{lname}_0)$.
(5) We now have a match for $\mathsf{InfAcc}Q$, so we have a successful proof.

Here is the generated **plan**:

(1) The firing of the accessibility axiom in the third step above generates access command $T_1 \Leftarrow \mathsf{mt}_{\mathsf{Udirect}} \Leftarrow \emptyset$, where T_1 is a table with attributes for eid_0 and lname_0.
(2) The accessibility axiom on the fourth line generates commands $T_2 \Leftarrow \mathsf{mt}_{\mathsf{Profinfo}} \Leftarrow \pi_{\mathsf{eid}_0} T_1$ and $T_2 := T_2 \bowtie T_1$.
(3) The match at the end generates the command output $\pi_\emptyset(T_2)$, which returns non-empty if T_2 is non-empty.

That is, we do an input-free access on $\mathsf{Udirect}$ and put all the results into $\mathsf{Profinfo}$.$\Box$

5.1. Full definition of SPJ-plan generation algorithm

In the first presentation of the algorithm we did not deal with several "corner cases" concerning these rule firings:

— the fact $F = R(c_1 \dots c_n)$ may contain not only constants that are produced in the chase proof ("chase constants") but also constants from the schema (e.g. "Smith", 3, etc.),
— the fact may have some chase constants repeated.

Here we complete the description to cover these cases.

We first discuss schema constants and repetition of chase constants in *input positions*. If $R(c_1 \dots c_n)$ has some chase constants repeated in an input position and possibly some of $c_{j_1} \dots c_{j_m}$ are schema constants, then in our input expression we perform a projection of T_{i-1} onto the attributes corresponding to the distinct chase constants in $c_{j_1} \dots c_{j_m}$ and then transform every tuple t into a corresponding tuple t' that can be used as an input to the access method mt. We do this transformation by repeating values or filling in positions with constants, using $R(c_1 \dots c_n)$ as a template. That is, if j_i is such that c_{j_i} is a chase constant c, then set t'_{j_i} to be c, while if c_{j_i} is a schema constant d, then set t'_{j_i} to be d. This transformation can be done with an SPJ query.

Example 5.2. Consider an accessibility axiom rule firing at step j of the form

$$\mathsf{accessible}(c_1) \wedge \mathsf{accessible}(\text{“Smith”}) \wedge R(c_1, c_1, \text{“Smith”}, c_2, c_2, \text{“Jones”}) \rightarrow \mathsf{InfAcc}R(c_1, c_1, \text{“Smith”}, c_2, c_2, \text{“Jones”})$$

Suppose that this rule firing was associated with access method mt on R having inputs on the first three positions of R. Note that the axioms guarantee that schema constants, like "Smith" are always in accessible.

If T_{i-1} is the temporary table produced by the commands associated with the $i-1^{th}$ firing, then T_{i-1} will have an attribute for c_1. We create an SPJ expression E_i that will take as input T_{i-1} and produce a table T_i with

$$\{(c_1 = v_1, c'_1 = v_1, c_{\text{“}\mathsf{Smith}\text{”}} = \text{“}\mathsf{Smith}\text{”}) | v_1 \in \pi_{c_1}(T_{i-1})\}$$

Such an E_i can be expressed using projection, a self-join, and a constant operator. We then generate an access command to mt using E_i as input. □

The modification of the algorithm that deals with repetition and schema constants in *output positions* (that is, non-input positions of the accessed relation) is similar, being done by post-processing, using a middleware query command to filter the output table down to those tuples that have constants in the proper positions and repeated values as found in the fact F. We then apply a projection to obtain a table whose attributes are the distinct chase constants in F.

Example 5.3. In the case of Example 5.2, the output table T'_i of the access command could have attributes $c_1, c'_1, c_{\mathsf{Smith}}, c_2, c'_2, c_{\mathsf{Jones}}$, and the output mapping would map the positions of R to these attributes. We then post-process by performing the middleware query command $T''_i = \pi_{c_1,c_2}\sigma_{c_2=c'_2 \wedge c_{\mathsf{Jones}}=\text{“}\mathsf{Jones}\text{”}}T'_i$. □

5.2. Properties of SPJ-plan generation and Proof of Theorem 4.4

We will now show that the prior algorithm proves Theorem 4.4: it takes any chase proof and produces an SPJ-plan that answers Q.

That is, we will prove:

PROPOSITION 5.4. *For every chase proof* $\mathsf{config}_1 \dots \mathsf{config}_k$ *proving* $\mathsf{InfAcc}Q$ *from* Q *the corresponding plan* PL *generated by Algorithm 1 answers* Q.

If config_i denotes the i^{th} configuration in the chase proof, then let $\mathsf{InfAccQuery}(\mathsf{config}_i)$ be the conjunctive query formed by taking the conjunction of all facts of the form $\mathsf{InfAcc}R(\vec{c})$ in config_i and turning them into an existentially quantified conjunction of facts $R(\vec{w})$, changing the chase constants c that satisfy $\mathsf{accessible}(c)$ to free variables and the other chase constants to existentially quantified variables. Note that if config_i has a match for $\mathsf{InfAcc}Q$, then $\mathsf{InfAccQuery}(\mathsf{config}_i)$ entails Q. Recall that our algorithm generated commands for every firing of an accessibility axiom, producing a corresponding plan PL^i, which produces a temporary table T_i. The attributes of T_i will be all chase constants in the interpretation of accessible within config_i — hence these match the output attributes of $\mathsf{InfAccQuery}(\mathsf{config}_i)$.

Given an instance I, a mapping from the chase constants present in the chase configuration config_i produced by the i^{th} step of the algorithm to I which preserves all facts in config_i within the original schema will be called a Sch config_i*-tuple of* I. As the notation implies, we will consider such elements as tuples with attributes from the constants of config_i.

We let $T_i(I)$ be the instance of table T_i produced by the plan PL^i when run on an instance I of schema Sch. Although the notation omits the dependence on PL^i, the value of T_i in PL^j, whenever it is well-defined, is independent of j.

We claim that the following "universality properties" hold for any instance I of Sch:

— For every Sch config_i-tuple of I, its projection on to those constants of config_i satisfying accessible is in $T_i(I)$.
— $T_i(I)$ is a subset of the tuples in $\mathsf{InfAccQuery}(\mathsf{config}_i)(I)$.

We explain why these two assertions together imply Proposition 5.4. First consider a tuple t returned by Q on a database instance I satisfying the Sch integrity constraints. I can be extended, just via duplicating its relations, to an instance I^* satisfying the accessible schema AcSch(Sch). Since t is returned by Q, there is a homomorphism h_1 of the canonical database of Q to t. In I^*, we can mimic each rule firing that produced the facts of config_i, and thus for each i we can extend h_1 to a mapping h_i that preserves all facts of config_i. Restating in the terminology of this section, for each config_i we can extend t to a Sch config_i-tuple t'. Now by the first assertion above, the projection of t' on the constants satisfying accessible is in $T_i(I)$. So, in particular, t, the projection of t' onto the constants corresponding to free variables of Q, is in the projection of $T_k(I)$, which is the final result of the top-level plan generated by the algorithm. Conversely, consider that for the final configuration config_k, $\mathsf{InfAccQuery}(\mathsf{config}_k)$ entails Q, as noted above. Thus by the second assertion, any tuple in the projection of $T_k(I)$ must satisfy Q on I.

The two assertions above are proven by induction on i.

We consider the inductive step for both assertions corresponding to an application of an accessibility axiom. We first give the argument for the basic version of the algorithm, ignoring repetition of variables and schema constants, and then discuss how to extend to incorporate the full version.

We consider the inductive case for firing on an accessibility axiom. Fix a Sch config_i-tuple t and let t_i be its projection on the constants satisfying the predicate accessible. By induction the projection of t on the accessible constants of config_{i-1} is in $T_{i-1}(I)$. In the absence of repetition of variables and constants, we know $T_i(I)$ is formed from $T_{i-1}(I)$ by "joining on the access": projecting tuples in $T_{i-1}(I)$ on the attributes corresponding to inputs to the access, performing the access, and joining the corresponding outputs to $T_{i-1}(I)$. We need to show that t extends some tuple returned by this join.

Assume that the accessibility axiom fired in the inductive step was associated with the exposure of some fact $R(c_1 \dots c_n)$, where $c_{j_1} \dots c_{j_m}$ satisfied accessible relations in the chase sequence up that point. Let t_{Inp} be the projection of t on these input attributes, and t' be the projection on all the $c_1 \dots c_n$. We claim that t' would be returned by an access on mt using t_{Inp}. This is clear, since for the corresponding accessibility axiom to fire, $R(c_1 \dots c_n)$ must hold in config_{i-1}, and thus the $c_1 \dots c_n$ attributes of a Sch config_i-tuple like t must satisfy R in I.

This completes the argument for the first assertion in the inductive case for firing an accessibility axiom, the argument under the assumption of no repetition of variables and no schema constants. In the general case, we let t_{Inp} be formed from t as above, but incorporating selections and repetition of variables on the input positions corresponding to the fact being exposed. Let t' be formed by taking a tuple in the accessed relation R consistent with t_{Inp} and applying operations enforcing repetitions and constants in the output positions. We again see that t' would be returned by the pre-processing, method access, and post-processing operations generated in the inductive step of the algorithm. This is because t is a Sch config_i-tuple and the corresponding fact added in config_i must have obeyed these selections in order for the corresponding accessibility axiom to fire.

We now turn to verifying the second assertion in the inductive case for the firing of an accessibility axiom. Consider an arbitrary database instance I satisfying the constraints of the schema, and let t_i be a tuple of $T_i(I)$. We show that t_i is returned by $\mathsf{InfAccQuery}(\mathsf{config}_i)$.

$\mathsf{InfAccQuery}(\mathsf{config}_i)$ is an existentially quantified conjunction. We first consider the basic case of the algorithm, with no repetition of variables or schema constants. In this case the axiom firing is of the form:

$$\mathsf{accessible}(c_{j_1}) \wedge \dots \mathsf{accessible}(c_{j_m}) \wedge R(c_1 \dots c_n) \rightarrow \\ \mathsf{InfAcc}R(c_1 \dots c_n) \wedge \bigwedge_j \mathsf{accessible}(c_j)$$

Recall that the access command produced by this rule firing would be

$$T'_i \Leftarrow \mathsf{mt} \Leftarrow \pi_{\{c_{j_1} \dots c_{j_m}\}}(T_{i-1})$$

where mt is an access method on R with input positions $j_1 \dots j_m$. To obtain $T_i(I)$, we joined T'_i with T_{i-1}.

By induction we know that t_{i-1}, the projection of t_i on the accessible constants of config_{i-1}, satisfies $\mathsf{InfAccQuery}(\mathsf{config}_{i-1})$, since t_{i-1} must be in $T_{i-1}(I)$. We consider the first interesting "new" conjunct being added at this stage, corresponding to the fact produced by the firing of an accessibility axiom. Since t_i is in $T_i(I)$, its projection to $\{c_1 \dots c_n\}$ must satisfy R. Therefore the atom corresponding to $R(c_1 \dots c_n)$ in $\mathsf{InfAccQuery}(\mathsf{config}_i)$ is satisfied by t_i. Thus we have proved the second assertion in the inductive step for an accessibility axiom, for the simplified version of the algorithm.

In the general case with constants and repetition, the argument is similar, but we argue that the pre- and post-postprocessing operations guarantee that t_i must reflect the repetitions and schema constants present in the fact exposed by the access.

We now turn to the inductive cases corresponding to the firing of Sch and $\mathsf{InfAccCopy}$ rules. Note that the first assertion is preserved by these rules, since the set of Sch config_i-tuples can only become smaller or stay the same, as we are required to preserved more facts.

For the second assertion, there is nothing to prove when a Sch rule is fired. In an inductive step for the $\mathsf{InfAccCopy}$ constraint rules, conjuncts are added to $\mathsf{InfAccQuery}(\mathsf{config}_i)$. These can easily be seen to hold because the instance I satisfies the integrity constraints of the schema.

This finishes the proof of the second assertion. It also completes the proof of Proposition 5.4.

We are now ready to give the proof of Theorem 4.4.

In one direction, suppose we have an SPJ-plan that answers query Q with respect to the constraints in Sch. Since SPJ-plans are monotone in the accessible data, Q is monotonically-access-determined. Thus by Claim 2, Q entails $\mathsf{InfAcc}Q$ with respect to the constraints in $\mathsf{AcSch}(\mathsf{Sch})$. By completeness of chase proofs, this means that there is a chase proof.

In the other direction, suppose that Q entails $\mathsf{InfAcc}Q$ with respect to the constraints in $\mathsf{AcSch}(\mathsf{Sch})$. By completeness of chase proof, we have a chase proof witnessing this. By Proposition 5.4 Algorithm 1 produces an SPJ-plan that answers Q.

This completes the proof of Theorem 4.4.

5.3. RA-plans for schemas with TGDs

We have proven Theorem 4.4 by giving a proof-to-plan algorithm that takes a proof that query Q is access-monotonically-determined, and produces an SPJ-plan. One can show that there are conjunctive queries and TGD constraints for which there is an RA-plan but no $USPJ$-plan. In fact, [Nash et al. 2010] shows that even in the case of constraints that define conjunctive query views, there are conjunctive queries that have reformulations over the views, but the reformulations require the relational difference operator. We will discuss a witness of this in Example 5.5 in the next section.

From this it follows that there are access schemas with TGD constraints and conjunctive queries that have RA-plans over the schema but no SPJ-plans. We now turn to devising an algorithm that proves Theorem 4.2, taking as input a chase proof using $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$, thus "proving that Q is access-determined", and outputting an RA-plan.

For ease of exposition we assume that our constraints contain no constants from the schema, and that our queries and constraints contain no repeated variables in atoms. Thus the chase proofs will not produce any configurations that contain such facts. The algorithm is generalized to the case where constants are present and there is repetition along the same lines as the algorithm for SPJ-plans, by introducing pre- and post-processing middleware query commands around access commands.

Algorithm Description. Our algorithm will proceed not by forward induction on proofs, as was the case with SPJ-plan generation, but by a *backward induction*. The algorithm takes as input a suffix $\mathsf{config}_i \ldots \mathsf{config}_j$ of a full proof consisting of chase configurations $\mathsf{config}_1 \ldots \mathsf{config}_j$, and produces a nested plan PL_i, where nested plans are as defined in Section 3. The plan PL_i generated from suffix $\mathsf{config}_i \ldots \mathsf{config}_j$ will include a distinguished temporary table $T_{\vec{x}^i}$ with attributes $\vec{x}^i$ that are the *accessible chase constants* in config_i, denoted $\mathsf{accessible}(\mathsf{config}_i)$: those that satisfy the predicate $\mathsf{accessible}$ in config_i. In the further inductive steps, PL_i will only be used in subplan calls with $T_{\vec{x}^i}$ being the table substituted. We will thus write $\mathsf{PL}_i(\vec{x}_i)$ to indicate that PL_i is a a nested plan with distinguished table $T_{\vec{x}^i}$, referring to $T_{\vec{x}^i}$ as the *parameter table* of the plan, and $\vec{x}^i$ as the *parameters*.

The output of the plan PL_i will be a table having attributes for chase constants that are either in $\mathsf{accessible}(\mathsf{config}_i)$ or which correspond to free variables of the query.

The algorithm proceeds by downward induction on i.

If $j = i$ (so only one configuration in the proof suffix), the algorithm produces the single command $\mathsf{Return}\ T_{\vec{x}^j}$. Note that in this case $\mathsf{accessible}(\mathsf{config}_j)$ must already contain the free variables of Q.

The pseudo-code for the inductive cases of the algorithm is listed in Figure 1. We can verify that the algorithm indeed returns tuples with attributes for all $\mathsf{accessible}(\mathsf{config}_i) \cup \mathsf{Free}(Q)$-constants. Thus for a full proof the output will have attributes corresponding to the free variables of the query Q. For a full proof, the set of parameters $\vec{x}^1$ is empty, since no attributes in the initial configuration satisfy $\mathsf{accessible}$. We take the top-level output of our plan-generation algorithm to be the result of sub-

— For Sch or $\mathsf{InfAccCopy}$ constraints, the algorithm just returns PL_{i+1}. Note that $\mathsf{accessible}(\mathsf{config}_i) = \mathsf{accessible}(\mathsf{config}_{i+1})$ in this case.

— We consider a suffix $\mathsf{config}_i \ldots \mathsf{config}_j$ where the transition from config_i to config_{i+1} is formed via a forward accessibility axiom firing via access mt exposing fact $R(\vec{c})$. We will generate the nested plan:

$$T_0^{i+1} \Leftarrow \mathsf{mt} \Leftarrow \pi_{c_{j_1} \ldots c_{j_m}} T_{\vec{x}^i}$$
$$T_1^{i+1} := T_0^{i+1} \bowtie T_{\vec{x}^i}$$
$$T_{i+1} \Leftarrow \mathsf{PL}_{i+1}[T_{\vec{x}^{i+1}}] \Leftarrow T_1^{i+1}$$
$$\text{Return } \pi_{\mathsf{Free}(Q) \cup \mathsf{accessible}(\mathsf{config}_i)} T_{i+1}$$

— We now consider a suffix $\mathsf{config}_i \ldots \mathsf{config}_j$ where the transition from config_i to config_{i+1} is formed via a backward accessibility axiom firing exposing fact $\mathsf{InfAcc}R(\vec{c})$. We will generate a plan the differs from the plan in the forward case by replacing the last line by commands returning empty if T_0^{i+1} is empty, and otherwise returning:

$$\{\vec{u} \in \pi_{\mathsf{Free}(Q) \cup \mathsf{accessible}(\mathsf{config}_i)}(T_{i+1}) \mid \exists \vec{t} \in T_{\vec{x}^i} \quad \vec{u} \in \bigcap_{\vec{w} \in T_1^{i+1} \, \pi_{\vec{x}^i} \vec{w} = \vec{t}} \pi_{\mathsf{Free}(Q) \cup \mathsf{accessible}(\mathsf{config}_i)}(\{\vec{z} \in T_{i+1} \mid \pi_{\mathsf{accessible}(\mathsf{config}_{i+1})} \vec{z} = \vec{w}\})\}$$

Although we have written the last expression in a mixture of relational algebra and logic, the generation of the last expression from T_{i+1} and T_1^{i+1} can be performed in relational algebra.

Fig. 1. Pseudo-code for generating RA-plans for schemas with TGDs

stituting for the parameter table the singleton instance with only the empty tuple. We denote this instance by $\emptyset$ below. Thus when the input is the maximal suffix consisting of a full proof starting from the canonical database of a query Q, we will produce an ordinary relational algebra plan without reference to $T_{\vec{x}^1}$, and one whose output constants will be exactly the free variables of Q.

We now give some intuition for the above steps. In the case of a forward accessibility axiom, we generate a nested plan which, given an instance of the parameter table $T_{\vec{x}^i}$ consisting of a single tuple $\vec{t}$, acts as follows: it does an access to R using the projection of $\vec{t}$ to the chase constants $c_{j_1} \ldots c_{j_m}$. For each result tuple $\vec{w}$ that joins with $\vec{t}$, the plan calls $\mathsf{PL}_{i+1}(\vec{w} \bowtie \vec{t})$, where the join is on the common attributes, and projects the results back to the constants in $\mathsf{accessible}(\mathsf{config}_i) \cup \mathsf{Free}(Q)$. Finally, the plan returns the union of all of these projections.

In the step for the backward accessibility axiom, our goal is to generate a plan that is similar, but performing an intersection rather than a union. That is, our plan should behave as follows, given an instance of $T_{\vec{x}^i}$ consisting of a single tuple $\vec{t}$: it does an access to R using the projection of $\vec{t}$ to the chase constants $c_{j_1} \ldots c_{j_m}$, then returns the intersection of the projections of the sets $\mathsf{PL}_{i+1}(\vec{w} \bowtie t)$ to the constants in $\mathsf{accessible}(\mathsf{config}_i) \cup \mathsf{Free}(Q)$, where $\vec{w}$ ranges over tuples in the result that join with t. We define the intersection to be empty if there are no such tuples.

In the electronic appendix, we will show that the nested plan generated by the algorithm answers the query Q. By Theorem 3.4 we can "flatten" the output to an ordinary RA-plan.

We can now complete the proof of Theorem 4.2, using the same argument as in Theorem 4.4. If Q has an RA-plan, then it is access-determined, and Claim 1 implies that Q entails $\mathsf{InfAcc}Q$ with respect to $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$. In the other direction, if the entailment holds, we get an RA-plan that answer Q using the algorithm described above.

Example 5.5. We consider a variant of an example due to Afrati [Afrati 2011]. Our base signature consists of a binary relation R.

We have views V_3 storing the set of pairs of nodes connected by a path of length 3, and V_4 storing the set of pairs connected by a path of length 4.

That is, we have constraints that are universal quantifications of the following rules, which give definitions for the view tables:

$$\begin{aligned} V_3(x, y) \rightarrow \exists x_2\ \exists x_3\ R(x, x_2) \wedge R(x_2, x_3) \wedge R(x_3, y) \\ R(x, x_2) \wedge R(x_2, x_3) \wedge R(x_3, y) \rightarrow V_3(x, y) \\ V_4(x, y) \rightarrow \exists x_2\ \exists x_3\ \exists x_4\ R(x, x_2) \wedge R(x_2, x_3) \wedge R(x_3, x_4) \wedge R(x_4, y) \\ R(x, x_2) \wedge R(x_2, x_3) \wedge R(x_3, x_4) \wedge R(x_4, y) \rightarrow V_4(x, y) \end{aligned}$$

Our query Q asks for all pairs x_1, x_6 such that x_1 reaches x_6 via a path of length 5.

Afrati showed that Q can be rewritten over the views as:

$$\exists y_5\ [V_4(x_1, y_5) \wedge \forall y_2\ V_3(y_2, y_5) \rightarrow V_4(y_2, x_6)]$$

Afrati also argued that Q is not monotone in the views: we can have two instances such that for each view table the second instance has all the facts of the first instance, but the query result over the second instance does not contain the query result over the first. Hence there can not be any $USPJ$-plan.

Consider an access schema having the constraints above, with the view relations having input-free access and the base tables have no access. We can derive an RA-plan equivalent to the above rewriting through our proof-based method.

The proof begins with the canonical database of query Q:

$$C_1 = \{R(x_1, x_2), R(x_2, x_3), R(x_3, x_4), R(x_4, x_5), R(x_5, x_6)\}$$

We then apply a chase step with the last constraint above, one part of the definition of V_4, obtaining configuration C_2 which adds the fact:

$$V_4(x_1, x_5)$$

We can now apply a "forward accessibility axiom" to obtain a configuration C_3 with the additional fact:

$$\mathsf{InfAcc}V_4(x_1, x_5)$$

We can now apply the copy of the third constraint above, to obtain configuration C_4 adding additional facts:

$$\mathsf{InfAcc}R(x_1, z_2), \mathsf{InfAcc}R(z_2, z_3), \mathsf{InfAcc}R(z_3, z_4), \mathsf{InfAcc}R(z_4, x_5)$$

From this we can apply the copy of the second constraint, to obtain configuration C_5 adding fact:

$$\mathsf{InfAcc}V_3(z_2, x_5)$$

Applying a "backward accessibility axiom" we obtain configuration C_6 with the fact:

$$V_3(z_2, x_5)$$

And then applying a chase step with the first constraint above leads to configuration C_7 adding facts:

$$R(z_2, w_3), R(w_3, w_4), R(w_4, x_5)$$

We can then apply the last constraint to get to configuration C_8 adding fact:

$$V_4(z_2, x_6)$$

After this we can apply another "forward accessibility axiom" to obtain configuration C_9 adding fact:

$$\mathsf{InfAcc}V_4(z_2, x_6)$$

Finally, we apply a copy of the third constraint, reaching configuration C_{10} with additional facts:

$$\mathsf{InfAcc}R(z_2, q_3), \mathsf{InfAcc}R(q_3, q_4), \mathsf{InfAcc}R(q_4, q_5), \mathsf{InfAcc}R(q_5, x_6)$$

We now have a match for $\mathsf{InfAcc}Q$.

Applying the proof-to-nested-RA-plan algorithm we will find that it generates a nested RA-plan that corresponds to this rewriting. We give the interesting steps in the inductive construction, ignoring steps that only call the next inductively-defined program.

— From the final configuration C_{10}, we generate a plan P_{10} that takes as input a table with tuples having values for x_1, x_5, x_6, and z_2 and simply returns the table.
— From the suffix of the proof beginning at configuration C_8 we generate P_8 which takes a table T_{x_1,z_2,x_5} and performs an access on V_4, naming the output results z_2, x_6 and joining them with T_{x_1,z_2,x_5}, returning a table with all joined tuples with attributes x_1, x_5, x_6 and z_2.
— From the suffix beginning with configuration C_5 we generate a plan P_5 that takes as input a table with tuples having attributes x_1 and x_5. P_5 performs an access on V_3 and selects all tuples (z_2, x_5) matching the input value for x_5, calls P_8 on each corresponding x_1, z_2, x_5 and intersects the projection of the results to x_1, x_5, x_6. Thus $P_5(x_1, x_5)$ returns $\bigcap_{z_2|V_3(z_2,x_5)} \{x_1, x_5, x_6 | V_4(z_2, x_6)\}$.
— From the suffix beginning with configuration C_2 we generate P_2 that has no input parameters. P_2 performs an access to V_4, calls P_5 on all the resulting tuples x_1, x_5, and then unions the projection of the results on to x_1, x_6.
The plan P_1 returned as the top-level result of the algorithm will be equal to P_2.
$\square$

5.4. Finite instances and tame constraints

Our results have provided a characterization of when a query can be answered (relative to TGD constraints) over all instances, in terms of proofs of certain entailments. We have also provided a procedure that converts a proof to a plan that answers the query over all instances. It is easy to show that these proof-based characterizations do not hold when the plans are required only to be correct over finite instances.

We will show that for "tame" constraints classes, answerability over all instances and answerability over finite instances coincide, and in particular the characterization theorems and algorithms relating answerability to entailment still hold in the finite. We also show that the existence of a plan of the appropriate type is decidable. Our proof-to-plan algorithms do not suffice to show this, since they depend upon having already found an appropriate proof.

We will first show these results below for constraints given by GTGDs. Recall from Section 2 that these are TGDs where all the variables occurring on the left appear in a single atom of the left. We will show that similar results hold for constraints that satisfy chase termination, such as the weakly acyclic constraints of [Deutsch et al. 2008]. Indeed, such results will hold for constraints in other fragments of first-order logic that have the finite model property, such as the Guarded Negation Fragment [Bárány et al. 2011].

THEOREM 5.6. *Let* Sch *be a schema whose constraints are* GTGD*s and let Q be a conjunctive query. Then if* PL *is a $USPJ$-plan that answers Q over finite instances, then* PL *answers Q over all instances.*

PROOF. We first give the argument when Q is a boolean query. Fix $USPJ$-plan PL for Q. Clearly there is an $USPJ$ query that holds true on an instance exactly when PL does. Consider the following assertion: for every instance I satisfying the constraints of Sch, I satisfies the query given by PL iff satisfies the query Q. The property is a pair of query containments of conjunctive queries with respect to GTGD constraints. Results of Bárány, Gottlob, and Otto [Bárány et al. 2010] imply that such containments have a decidable satisfiability property, and have the finite model property:

if such a containment holds on all finite instances, it holds on all instances.

Hence PL answers Q over all instances.

The extension to the non-boolean case can be done by converting the free variables of Q into constants. □

Note that Theorem 5.6 implies that statements made about our proof-to-plan algorithms for SPJ plans over constraints consisting of GTGDs or TGDs with terminating chase within this paper still hold if plans are only required to answer a query over finite instances.

In the case of RA-plans, the situation is more complex. First, the analog of Theorem 5.6 fails: we can not say that for any RA-plan PL that answers a boolean query Q over finite instances, PL answers it over all instances. For example, an RA-plan may return true on all finite instances (and hence answer a tautological query Q) but return false on some infinite instance. However, for GTGDs, we can transfer between the existence of an RA-plan over finite instances and the existence arbitrary instances, and can still decide if this property holds.

THEOREM 5.7. *Let* Sch *be a schema whose constraints are* GTGD*s and let Q be a conjunctive query. Then there is an RA-plan that answers Q over finite instances iff there is an RA-plan that answers Q over all instances. Furthermore, we can decide whether this property holds.*

PROOF. Suppose that Q has an RA-plan that works over finite instances, consisting of k access commands. Consider the following property of two instances I and I' for schema Sch: I and I' both satisfy the constraints of the schema, they agree on every fact that can be extracted via at most k successive accesses (k iterations of the fixpoint process used to generate the accessible part), but they disagree on the truth value of $Q(\vec{c})$ for some $\vec{c}$.

The property can be expressed as a boolean combination of conjunctive queries and GTGD constraints. But it is easy to see that the output of a plan consisting of k access command depends only on the data that can be returned by k iterations of the fixpoint. Thus γ has no finite instance satisfying it. Since γ is unsatisfiable over finite instances, it is unsatisfiable over all instances (by [Bárány et al. 2010] again). Hence Q is access-determined over all instances. From Claim 1, we conclude that Q entails $\mathsf{InfAcc}Q$ over all instances, and hence by Theorem 4.2 Q has an RA-plan over all instances.

Decidability follows since whether Q entails $\mathsf{InfAcc}Q$ over all instances is a containment between CQs with respect to GTGD constraints, hence decidable by the results of [Bárány et al. 2010]. □

We now turn to decidability of questions related to RA-plans for TGDs with terminating chase. Unlike AcSch, the bidirectional schema $\mathsf{AcSch}^{\leftrightarrow}$ does not preserve the terminating chase property. However, one can check that for many well-known classes with terminating chase, such as weakly acyclic TGDs, the schema $\mathsf{AcSch}^{\leftrightarrow}$ is also in the same class. Hence the analog of Theorem 5.7 holds for such classes as well.

6. Low-cost plans via proof search

We have seen that proofs of an entailment can lead us to *some* successful plan whenever one exists.

We now look at finding efficient plans, focusing for the remainder of the paper on generating SPJ-plans with respect to schemas consisting of TGDs.

Example 6.1. We return to the variant of Example 1.3 mentioned in the introduction. We have a $\mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \mathsf{lname})$ datasource with one access method $\mathsf{mt}_{\mathsf{Profinfo}}$ requiring an eid as an input. There are also tables $\mathsf{Udirectory}_1(\mathsf{eid}, \mathsf{lname})$ with input-free access method mt_1 and $\mathsf{Udirectory}_2(\mathsf{eid}, \mathsf{lname})$ with input-free access method mt_2. The constraints include two inclusion dependencies:

$$\forall \mathsf{eid}\ \forall \mathsf{onum}\ \forall \mathsf{lname}\ \mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \mathsf{lname}) \rightarrow \mathsf{Udirectory}_i(\mathsf{eid}, \mathsf{lname})$$

for $i = 1, 2$. Our goal is to generate an SPJ-plan for query $Q = \mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \text{“Smith”})$.

The corresponding auxiliary schema will add tables $\mathsf{InfAccUdirectory}_1$, $\mathsf{InfAccUdirectory}_2$, and $\mathsf{InfAccProfinfo}$. The axioms will be the Sch constraints, $\mathsf{InfAccCopy}$ constraints, and “accessibility axioms” that include the following axioms, labelled (UA_i), for $i = 1, 2$:

$$\forall \mathsf{eid}\ \forall \mathsf{lname}\ \mathsf{Udirectory}_i(\mathsf{eid}, \mathsf{lname}) \rightarrow$$
$$\mathsf{InfAccUdirectory}_i(\mathsf{eid}, \mathsf{onum}, \mathsf{lname}) \wedge \mathsf{accessible}(\mathsf{eid}) \wedge \mathsf{accessible}(\mathsf{lname})$$

Also included will be the accessibility axiom (PA)

$$\forall \mathsf{eid}\ \forall \mathsf{onum}\ \forall \mathsf{lname}\ \mathsf{accessible}(\mathsf{eid}) \wedge \mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \mathsf{lname}) \rightarrow$$
$$\mathsf{InfAccProfinfo}(\mathsf{eid}, \mathsf{onum}, \mathsf{lname}) \wedge \mathsf{accessible}(\mathsf{onum}) \wedge \mathsf{accessible}(\mathsf{lname})$$

and the ground accessibility axiom:

$$\mathsf{accessible}(\text{“Smith”})$$

A chase proof will begin with the canonical database of Q, namely:

$$\mathsf{Profinfo}(\mathsf{eid}_0, \mathsf{onum}_0, \text{“Smith”})$$

Our general strategy will be that before looking for accessibility axioms to fire, we look first for any non-accessibility axioms we can apply first. Thus we apply the two original inclusion dependencies to add facts:

$$\mathsf{Udirectory}_2(\mathsf{eid}_0, \mathsf{onum}_0, \mathsf{lname}_0), \mathsf{Udirectory}_2(\mathsf{eid}_0, \mathsf{onum}_0, \mathsf{lname}_0)$$

We also fire the axiom generating fact $\mathsf{accessible}(\text{“Smith”})$, representing the fact that “Smith” is a known constant. If we model this accessibility axiom as corresponding to a special kind of “access method” it would clearly be a method with no cost.

Continuing from this are several possible proofs. One proof will apply a chase step with the rule (UA_1) to add facts

$$\mathsf{InfAccUdirectory}_1(\mathsf{eid}_0, \mathsf{onum}_0, \text{“Smith”}), \mathsf{accessible}(\mathsf{eid}_0)\mathsf{accessible}(\mathsf{onum}_0)$$

followed by a chase step with (PA) to add fact

$$\mathsf{InfAccProfinfo}(\mathsf{eid}_0, \mathsf{onum}_0, \text{“Smith”})$$

At this point we have a match for $\mathsf{InfAcc}Q$, and hence a complete chase proof.

Instantiating the TGD-based plan-generation algorithm (as in Figure 1, modified as in Subsection 5.1 to account for constants) we will generate the plan:

$T_1 \Leftarrow \mathsf{mt}_1 \Leftarrow \emptyset$
$T_2 := \sigma_{\mathsf{lname}=\text{“}\mathsf{Smith}\text{”}} T_1$
$T_3 \Leftarrow \mathsf{mt}_{\mathsf{Profinfo}} \Leftarrow \pi_{\mathsf{eid}}(T_2)$
Return $\pi_{\mathsf{onum}}(T_3)$

A second proof would be similar, but using (UA_2) rather than (UA_1). This would generate a plan that corresponded to an access to $\mathsf{Udirect}_2$ rather than $\mathsf{Udirect}_1$.

A third proof would first fire (UA_1) and then (UA_2), followed by (PA). Applying the TGD-based algorithm would automatically generate the plan:

$T_1 \Leftarrow \mathsf{mt}_1 \Leftarrow \emptyset$
$T_2 := \sigma_{\mathsf{lname}=\text{“}\mathsf{Smith}\text{”}} T_1$
$T_3 \Leftarrow \mathsf{mt}_2 \Leftarrow \emptyset$
$T_4 := \sigma_{\mathsf{lname}=\text{“}\mathsf{Smith}\text{”}} T_3$
$T_5 := T_2 \bowtie T_4$
$T_6 \Leftarrow \mathsf{mt}_{\mathsf{Profinfo}} \Leftarrow \pi_{\mathsf{eid}}(T_5)$
Return $\pi_{\mathsf{onum}}(T_5)$

What we see is that each “interesting plan” is captured by a distinct proof. Which of these plans has the lowest cost will depend on, e.g., the relative efficiency of the access methods mt_1 and mt_2, along with the amount of tuples each returns. Thus we can not make a decision on which plan is most efficient simply by looking at the proof. What we can do is explore this space of proofs while measuring the efficiency of the corresponding plans.□

How good are proof-based plans?. We now consider our first question about cost. Are proof-based plans as “cheap” as general plans?

We first consider a cost comparison between plans and conjunctive queries.

Recall from Section 3 that a conjunctive query Q with atoms $A_1 \dots A_n$ is *executable* relative to a schema with access patterns [Li and Chang 2000] if there is an annotation of each atom $A_i = R_i(\vec{x}_i)$ with an access method mt_i on R such that for each variable x of Q, for the first A_i containing x, x occurs only in an output position of mt_i. Proposition 3.1 provides a function that converts every executable **CQ** Q to an SPJ-plan $\mathsf{PlanOf}(Q)$ such that the number of accesses equals the number of atoms in Q, and conversely a function taking a plan PL to an executable query $\mathsf{CQOf}(\mathsf{PL})$ such that the number of atoms in the query equals the number of accesses in PL.

The following proposition shows that proof-based plans perform as well as executable CQs, for any notion of cost that is based on the set of methods called.

PROPOSITION 6.2. *For every **CQ** Q, schema Sch, and executable query Q' equivalent to Q there is a proof v such that if PL^v is the SPJ-plan produced by the proof-to-plan algorithm, then $\mathsf{CQOf}(\mathsf{PL}^v)$ has at most the number of atoms as Q', and PL^v uses no more methods than $\mathsf{PlanOf}(Q')$.*

In particular the cost of PL^v will be no more than that of $\mathsf{PlanOf}(Q')$ under any simple cost function.

PROOF. Let Q' be an executable query as above, with atoms $A_1 \dots A_n$, $A_i = R_i(\vec{x}_i)$, and let Q'_i be the subquery conjoining atoms $A_1 \dots A_i$. Let config_∞ be a (possibly infinite) instance resulting from chasing the canonical database of Q with the constraints of Sch. Q has a match on the elements in config_∞ corresponding to free variables of Q, and since Q' is equivalent to Q on instances satisfying the constraints, Q' must have such a match on config_∞ as well.

There is thus a finite subinstance config_0 of config_∞ on which Q' returns the elements corresponding to the free variables of Q. Although Q' consisted of n atoms, when evaluating $\mathsf{PlanOf}(Q')$ on config_0 we may have exposed many more than n facts. We can choose a single tuple F_i matching atom A_i so that there is a match of Q' on $F_1 \ldots F_n$.

We begin our chase proof by building config_0 and then firing the accessibility axioms that correspond to expose each F_i. Using the fact that Q' was executable, we can see that after exposing $F_1 \ldots F_{i-1}$, all values occurring in F_i within input positions of method mt_i will satisfy $\mathsf{accessible}$. Note that the facts $F_1 \ldots F_n$ may not be distinct, in which case it will be unnecessary to fire n accessibility axioms to expose $F_1 \ldots F_n$.

We now argue that we can complete the chase sequence into a proof by firing $\mathsf{InfAccCopy}$ rules. Consider the instance consisting of facts $F_1 \ldots F_n$, with each relation R renamed to $\mathsf{InfAcc}R$, and let config'_∞ result from chasing this instance with the $\mathsf{InfAccCopy}$ rules. config'_∞ clearly satisfies all the $\mathsf{InfAccCopy}$ constraints. Since $F_1 \ldots F_n$ led to a match for Q', the query $\mathsf{InfAcc}Q'$ has a match on config'_∞. As Q and Q' are equivalent on instances satisfying the Sch constraints, $\mathsf{InfAcc}Q'$ and $\mathsf{InfAcc}Q$ are equivalent on instances satisfying the copy of the constraints. Therefore there is a finite subinstance config'_0 of config'_∞ containing a match for $\mathsf{InfAcc}Q$. We complete our chase proof by firing $\mathsf{InfAccCopy}$ rules to generate config'_0. Because the resulting chase sequence v consists of firing Sch constraints, accessibility axioms, and $\mathsf{InfAccCopy}$ constraints, beginning with the canonical database of Q and leading to a configuration with a match for $\mathsf{InfAcc}Q$, it represents a chase proof of our entailment. Therefore the corresponding plan PL^v is equivalent to Q. The access commands in PL^v correspond to the accessibility axioms needed to expose $F_1 \ldots F_n$, which will in turn be a subsequence of the method sequence used in $\mathsf{PlanOf}(Q')$. $\square$

A similar argument shows that any plan that has a "left-deep" structure, joining on one access at a time, there is a proof-based plan that uses no more methods than it.

Example 6.3. The following example shows a case where proof-based plans are not as efficient as general plans.

Consider the query

$$Q = \exists xyz\ S(x) \wedge S(y) \wedge R(x, w) \wedge R(y, z) \wedge U(w) \wedge V(z)$$

and a schema where

— there is an input-free access mt_S on S
— there is an access method mt_R accessing R on the first position
— there are access methods mt_U and mt_V requiring the sole position on U and V, respectively

This is an executable CQ with 6 atoms. However, one can obtain the following plan PL to answer Q which uses only 4 access commands:

$$\begin{aligned}
U_1(x) &\Leftarrow \mathsf{mt}_S \Leftarrow \emptyset \\
U_2(y) &:= \text{ rename } U_1 \\
U_3(x, w) &\Leftarrow \mathsf{mt}_R \Leftarrow U_2 \\
U_4(y, z) &:= \text{ rename } U_3 \\
U_5(w) &\Leftarrow \mathsf{mt}_U \Leftarrow \pi_w(U_3) \\
U_6(z) &\Leftarrow \mathsf{mt}_V \Leftarrow \pi_z(U_4) \\
\text{Return } & U_1 \bowtie U_2 \bowtie U_3 \bowtie U_4 \bowtie U_5 \bowtie U_6
\end{aligned}$$

Above the renaming operations are syntactic sugar to make the variables clearer. The plan first accesses S to get possible values of x and y, then uses the resulting values in R. The output is then put in U and in V, and then the results are stitched together using a join. The plan answers Q.

Note that this plan does not have the "left-deep" shape produced by either proof-based plans or the naïve translation of executable queries. And indeed there is no proof-based plan that generates a plan with this number of access commands.

Let the initial configuration of the chase contain

$$\{S(x_0), S(y_0), R(x_0, w_0), R(y_0, z_0), U(w_0), V(z_0)\}$$

A chase-based proof along the lines above would proceed via the following rule firings:

— $S(x_0) \rightarrow \mathsf{InfAcc}S(x_0)$
— $S(y_0) \rightarrow \mathsf{InfAcc}S(y_0)$ as well as generating $\mathsf{accessible}(x_0)$, $\mathsf{accessible}(y_0)$.
— $R(x_0, w_0) \wedge \mathsf{accessible}(x_0) \rightarrow \mathsf{InfAcc}R(x_0, w_0)$
— $R(y_0, z_0) \wedge \mathsf{accessible}(x_0) \rightarrow \mathsf{InfAcc}R(y_0, z_0)$
— $U(w_0) \wedge \mathsf{accessible}(w_0) \rightarrow \mathsf{InfAcc}U(w_0)$
— $V(z_0) \wedge \mathsf{accessible}(z_0) \rightarrow \mathsf{InfAcc}V(z_0)$

These rules generate inferred accessible facts that match $\mathsf{InfAcc}Q$. Note that there are two rule firings on relation R.

If we put the proof sequence above into our plan-generation algorithm, we get a plan that will have 6 access commands, one for each firing. The plan PL' will still generate two calls to access method R, unlike the one above. These calls will use the same inputs, and in an intelligent wrapper that caches the results of prior accesses made, the second call will require no tuples. Related observations about the superiority of "bushy-plans" to left-deep plans in the presence of access restrictions have been known for some time: see, for example, [Florescu et al. 1999] Example 3.2.

One can see that the two plans generate exactly the same concrete accesses. But the number of "bulk method calls" is larger in PL', hence a cost function that counts the number of calls will give a higher cost to PL' than to PL. □

We now compare proof-based plans and general (not necessarily left-deep) plans in terms of the set of accesses made at runtime. An SPJ-plan PL *uses no more runtime accesses than* SPJ-plan PL', denoted $\mathsf{PL} \preceq_{\mathsf{RTA}} \mathsf{PL}'$ if for every pair consisting of a method mt and method input $\vec{t}$ that is executed in running PL on instance I of the schema, the same pair is also executed in running PL' on I.

We show that proof-based plans are optimal with respect to arbitrary plans when runtime accesses are considered:

THEOREM 6.4. *For conjunctive query Q and access schema with TGD constraints Σ, for every SPJ-plan PL that answers Q, there is a chase sequence v proving $\mathsf{InfAcc}Q$, such that, letting PL^v be the SPJ-plan PL^v generated from v via the proof-to-plan algorithm,* $\mathsf{PL}^v \preceq_{\mathsf{RTA}} \mathsf{PL}$.

Note that this theorem does not imply anything about the cost of proof-based plans versus arbitrary plans according to particular cost functions, since cost functions look at plans statically, and are thus not necessarily monotone in the set of (method, input) pairs produced at runtime.

The proof of this result is deferred to an electronic appendix available via the ACM Digital Library.

Summarizing, if we are interested only in the number of accesses generated at runtime, it suffices to look at proof-based plans. In addition, if we measure cost via some function of the set of access commands (ignoring middleware cost), then proof-based plans are as good as arbitrary "left-deep" plans. Although most realistic cost functions would consider more than just the set of commands, we take this as a rough justification for restricting to proof-based plans.

6.1. Simultaneous proof and plan search

We now turn to algorithms that search for a low-cost proof-based plan.

Note that in the algorithm that generated SPJ-plans from proofs, the plans were generated inductively on the number of steps in a prefix of a proof. Consequently, we can associate a partial plan to a partial proof. This allows us to measure the cost of the corresponding partial plan during proof exploration. These two observations underlie the idea that *we can find low-cost plans by exploring the space of proofs*.

Our search will maintain a *partial proof tree* – a tree consisting of chase sequences, ordered by extension. We refer to the configuration of the final element in the chase sequence associated with a node v as $\mathsf{config}(v)$ The plan associated with v is the one generated by the proof-to-plan algorithm given previously, while by the cost of v we mean the cost of the associated plan. We now give an algorithm for extending the tree to find new proofs.

Definition 6.5 (*Candidate facts for exposure*). Consider a node v such that there is a fact $R(c_1 \dots c_m)$ in $\mathsf{config}(v)$ with $\mathsf{InfAcc}R(c_1 \dots c_m)$ not yet in $\mathsf{config}(v)$ and there is an access method $\mathtt{mt}$ on R with input positions $j_1 \dots j_m$ such that $\mathsf{accessible}(c_{j_1}) \dots \mathsf{accessible}(c_{j_m})$ all hold in $\mathsf{config}(v)$. Then we call $R(c_1 \dots c_m)$ a candidate for exposure at v, and $\mathtt{mt}$ an *exposing method for* $R(c_1 \dots c_m)$.

Note that if a fact is a candidate for exposure, then firing an accessibility axiom will add that fact to the associated chase sequence.

When we explore the impact of making an access, we want to include all relevant consequences that do not involve further accesses, thus producing an *eager proof* which corresponds to the following requirements on the configurations in a partial proof tree:

— (Original Schema Reasoning First) The configuration of the root node (henceforward "initial configuration") corresponds to the canonical instance of Q plus the result of firing of a sufficient sequence of Sch integrity constraints until a termination condition is reached — the notion of sufficient sequence will be explained further below.
— (Fire Inferred Accessible Rules Immediately) For a non-root node v, there is a candidate fact for exposure $R(c_1 \dots c_m)$ in its parent with exposing method $\mathtt{mt}$ such that $\mathsf{config}(v)$ is obtained from the parent by
  — adding the *fact induced by firing* $\mathtt{mt}$ *with* $c_{j_1} \dots c_{j_m}$, namely $\mathsf{InfAcc}R(d_1 \dots d_m)$.
  — firing a sufficient set of inferred accessible axioms on the result.

Thus the successive configurations are connected by firing a rule associated with an accessibility axiom and exploring the "cost-free" consequences. Thus we can also characterize a node v by the sequence of rule firings of accessibility axioms leading to it. It is easy to see that an arbitrary proof can be converted into an eager proof via reordering.

We also label a node as *successful* if $\mathsf{InfAcc}Q$ holds in the corresponding configuration (preserving free variables in the non-boolean case).

The idea is that we have labelled each node with a configuration of the proof, and whenever we choose an accessibility axiom to fire, after firing we immediately fire all the relevant rules that do not generate accesses.

We explore downward from a node v of a partial proof tree by choosing a *candidate fact for exposure* at $\mathsf{config}(v)$ along with the methods that expose the fact. A node is *terminal* if it is either successful or has no candidate facts. Note that non-terminal nodes

Algorithm 2: plan search

Input: query Q, schema S
Output: plan BestPlan

1 ProofTree := an initial node v_0 labelled with the configuration obtained by a sufficient number of firings of Sch constraints.
2 Set Candidates(v_0) = all pairs $(R(c_1 \dots c_n), \mathsf{mt})$ with $R(c_1 \dots c_n)$ a fact in the original configuration and mt a method on R.
3 BestPlan := $\perp$
4 BestCost := ∞
5 **while** *there is a non-terminal node* $v \in$ ProofTree **do**
6   Choose such a node v.
7   Choose a candidate fact and method $(R(c_1 \dots c_n), \mathsf{mt}) \in \mathsf{Candidates}(v)$ with $\mathsf{accessible}(c_{j_1}) \dots \mathsf{accessible}(c_{j_m}) \in \mathsf{config}(v)$ and mt having inputs $j_1 \dots j_m$.
8   Add a new node v' as a child of v with configuration formed by adding $\mathsf{InfAcc}R(c_1 \dots c_n)$ a sufficiently large closure by firings of the InfAccCopy constraints.
9   Remove $(R(c_1 \dots c_n), \mathsf{mt})$ from Candidates(v), marking v as terminal if it has no more candidates.
10   Determine if v' is successful by checking if $\mathsf{InfAcc}Q$ holds, and if so also mark it as terminal.
11   **if** v' *is successful and* $\mathsf{Cost}(\mathsf{Plan}(v')) < \mathsf{BestCost}$ **then**
12   BestPlan := Plan(v')
13   BestCost := Cost(Plan(v'))
14 return BestPlan;

do not have to be leaves of the tree. We continue until a sufficient set of accessibility axioms have been fired.

The basic search structure is outlined in Algorithm 2. At each iteration of the while loop at line 5 we have a partial proof/plan tree satisfying the properties above. We look for a node v corresponding to a partial proof that is not yet successful, and for which the firings of accessibility axioms can add new facts. We non-deterministically choose such a path and such an axiom (lines 6-7), and calculate the new configuration that comes from firing the rule, along with the commands that will be added to the corresponding plan (line 8). We update the candidate list (line 9) and determine whether the new path is successful, recording whether this gives the new lowest cost plan (line 11).

Termination. In Algorithm 2 there are several points where we limit the search to achieve termination. Formally, we need: (1) A finite sequence v_0 formed by closing the canonical query Q under firings of Sch rules. We use this set in the step of chasing with the Sch constraints in line 1. Given v_0, we have a bound on termination of the while loop of line 5, since we only expose facts from v_0. (2) For each chase sequence w_0, an extension $v'(w_0)$ of w_0 by firing InfAccCopy constraints, used within every step of the while loop on line 8.

For classes with terminating chase, we set v_0 to be any set of firings of Sch rules such that there are no active triggers of Sch constraints. We let $v'(w)$ to be any extension of w by InfAccCopy constraints with no active triggers among InfAccCopy constraints.

We can show that this algorithm will solve the low-cost plan problem for proof-based plans.

THEOREM 6.6. *Consider any simple cost function* Cost*, access schema* Sch *whose constraints are TGDs with terminating chase, and conjunctive query* Q*. Then Algorithm 2, instantiated with the sufficient sets above and the cost function* Cost*, will al-*

ways return a plan with the lowest cost among all those proof-based plans that completely answer Q w.r.t. Sch*, or return $\perp$ if there is no plan that answers Q.*

PROOF. Consider any chase proof w that Q entails $\mathsf{InfAcc}Q$. Such a w can be extended so that there are no active triggers among Sch or InfAccCopy constraints, without changing the resulting plan. Let PL^w be the plan and $\mathsf{mt}_1 \ldots \mathsf{mt}_n$ the sequence of access methods within the access commands of PL^w. Let v_0 be the chase sequence described by chasing the canonical database with Sch rules, as described above, and config_0 be the final chase configuration in v_0. When PL^w is run on config_0, it will execute access commands $\mathsf{Command}'_1 \ldots \mathsf{Command}'_n$ with inputs $I_1 \ldots I_n$, with the output of the access being $O_1 \ldots O_n$. As in the proof of Proposition 6.2, we can find facts $F_1 \ldots F_n$ with corresponding tuples $\vec{t}_1 \in O_1 \ldots \vec{t}_n \in O_n$ accessed by PL^w on config_0 such that these outputs suffice to return the tuple corresponding to the free variables of Q. Let v' be the chase sequence corresponding to firings of accessibility axioms exposing $F_1 \ldots F_n$ on config_0. Arguing as in the proof of Proposition 6.2, we can see that this represents a valid sequence of firings, since the values of $\vec{t}_i$ lying within the input positions of mt_i will satisfy accessible. Arguing as in Proposition 6.2 again, we see that $v_0 \cdot v'$ can be extended by the firing of some collection of InfAccCopy constraints v_3, giving a proof of the entailment of $\mathsf{InfAcc}Q$ from Q. Letting $v = v_0 \cdot v' \cdot v_3$, we claim that the corresponding plan PL^v will be discovered by the algorithm. Since PL^v uses no more methods than PL^w, and our cost function is simple, the cost of PL^v will be no higher than that of PL^w. Thus if PL^v is discovered by the algorithm, we have proven optimality.

Consider the chase sequence v^* formed from $v_0 \cdot v' \cdot v_3$ by removing v_3 and inserting after each prefix p_3 of $v_0 \cdot v'$ a chase sequence of InfAccCopy constraints that are applicable after that prefix, until there are no InfAccCopy constraints that can apply. PL^{v^*} and PL^v are the same, since the accessibility axiom firings in both sequences are identical. We can also see that v^* will have a match for $\mathsf{InfAcc}Q$. This holds because both v^* and v extend the sequence $v_0 \cdot v'$ by firing InfAccCopy constraints until there are no active triggers by InfAccCopy constraints, hence they both satisfy exactly the CQs that are implied by the final configuration of $v_0 \cdot v'$ and the InfAccCopy constraints.

We argue that in every iteration of the while loop (i) if v^* has not been discovered then the while loop at line 5 will not yet terminate, (ii) v^* will always have a prefix in ProofTree with a non-empty set of candidates. Since candidates are removed in each iteration of the while loop, eventually such an ancestor prefix will be chosen to be expanded, and thus v^* will be discovered. $\square$

Such sufficient sets also exist for classes without terminating chase, such as GTGDs. Instead of firing the rules until termination, it is enough to fire them a sufficient number of times. The details of this result are given in the online appendix.

Example 6.7. Let us return to the setting of Example 1.3, assuming we have three directory sources $\mathsf{Udirect}_1, \mathsf{Udirect}_2, \mathsf{Udirect}_3$. The integrity constraints contain:

$$\mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \mathsf{lname}) \rightarrow \mathsf{Udirect}_i(\mathsf{eid}, \mathsf{lname})$$

for $i = 1, 2, 3$, with Profinfo having an access that requires the first argument to be given and each $\mathsf{Udirectory}_i$ having unrestricted access. Consider the query $Q = \exists \mathsf{eid}\ \mathsf{onum}\ \mathsf{lname}\ \mathsf{Profinfo}(\mathsf{eid}, \mathsf{onum}, \mathsf{lname})$.

The behavior of our exploration is illustrated in Figure 2. The canonical database of Q consists of the fact $\mathsf{Profinfo}(\mathsf{eid}_0, \mathsf{onum}_0, \mathsf{lname}_0)$. The configuration of the initial node n_0 will then add $\mathsf{Udirect}_i(\mathsf{eid}_0, \mathsf{lname}_0)$ for $i = 1, 2, 3$. There are thus three candidates facts to expose, $\mathsf{Udirect}_i(\mathsf{eid}_0, \mathsf{lname}_0) : i = 1, 2, 3$ in the initial node.

We might choose $\mathsf{Udirect}_1(\mathsf{eid}_0, \mathsf{lname}_0)$ to expose first. This creates a child n_1 with transition from parent to child associated with an access on $\mathsf{Udirect}_1$,

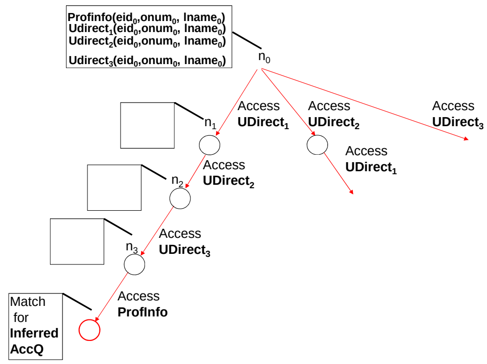

Fig. 2. Exploration in the running example

putting the output into a table T_1 with attributes $\{\mathsf{eid}_0, \mathsf{lname}_0\}$. The configuration for n_1 adds the fact $\mathsf{InfAccUdirect}_1(\mathsf{eid}_0, \mathsf{lname}_0)$, and then immediately the fact $\mathsf{InfAccUdirect}_1(\mathsf{eid}_0, \mathsf{lname}_0), \mathsf{accessible}(eid_0)$, and $\mathsf{accessible}(\mathsf{lname}_0)$.

In n_1 there are three candidates to expose: $\mathsf{Udirect}_2(\mathsf{eid}_0, \mathsf{lname}_0), \mathsf{Udirect}_3(\mathsf{eid}_0, \mathsf{lname}_0)$ and now also $\mathsf{Profinfo}(\mathsf{eid}_0, \mathsf{onum}_0, \mathsf{lname}_0)$, since there is an accessibility axiom that would expose this last fact now. Again, based on some ordering policy we might choose $\mathsf{Udirect}_2(\mathsf{eid}_0, \mathsf{lname}_0)$ to expose, and a child n_2 will be generated (again including the exposed fact $\mathsf{InfAccUdirect}_2(\mathsf{eid}_0, \mathsf{lname}_0)$ and one inferred fact). The transition to n_2 will be associated with an access command on $\mathsf{Udirect}_2$ and a command joining the results with the previous table.

The node n_2 will have two candidates to expose, $\mathsf{Udirect}_3(\mathsf{eid}_0, \mathsf{lname}_0)$ and $\mathsf{Profinfo}(\mathsf{eid}_0, \mathsf{onum}_0, \mathsf{lname}_0)$. Assuming our policy chooses to expose $\mathsf{Udirect}_3(\mathsf{eid}_0, \mathsf{lname}_0)$ next, we will generate a child n_3, whose configuration adds the exposed fact and the corresponding inferred accessible fact.

The node n_3 will have only one candidate fact, corresponding to $\mathsf{Profinfo}(\mathsf{eid}_0, \mathsf{onum}_0, \mathsf{lname}_0)$. Selecting this fact a child n_4 will be generated. The access associated with the addition of n_4 will be of the form $T_4 \Leftarrow \mathsf{mt}_{\mathsf{Profinfo}} \Leftarrow T_3$, where T_3 will be a table with attributes $\mathsf{eid}_0, \mathsf{lname}_0$ containing the intersection of the outputs of the 3 prior accesses. The query $\mathsf{InfAcc}Q$ matches the configuration of n_4, so it is designated a success node, hence is a leaf in the search.

Now the search can go back up the tree to a node with more candidates to explore — e.g. it might move to pick a child of n_3 to explore.

Note that at some point in the process, the extension process will consider creating a node n''' corresponding to the sequence of commands $T_1 \Leftarrow \mathsf{mt}_{\mathsf{Udirect}_2} \Leftarrow \emptyset; T_2 \Leftarrow \mathsf{mt}_{\mathsf{Udirect}_1} \Leftarrow \emptyset$, and continuing on this way will explore all left-deep join trees between the $\mathsf{Udirect}_i$ datasources. □

7. Conclusions and Future Work

The main goal of this work is to introduce a means for generating query plans from proofs that a certain semantic property holds for the query. We show that the proof goals can be modified for different kinds of target plans. We give some evidence that the plans that are generated by proofs are as good as general plans under some metrics, and show that exploring many proofs, one can arrive at optimal proof-based plans.

In follow-on work, we discuss implementation of a system based on generating plans from proofs [Benedikt et al. 2014; 2015]. In current work we are extending the system to more general constraints beyond TGDs, using a tableau-based proof system.

8. Related Work

Work on querying in the presence of access patterns without integrity constraints. Work on access patterns was initially motivated by the goal of finding index-only plans over a fixed set of indices in traditional databases [Ullman 1989; Rajaraman et al. 1995]. Li and Chang [Li and Chang 2001; Li 2003] later explored the complexity of determining when a query could be answered in the presence of access patterns, where answering the query referred to coming up with an executable query. Extensions to richer queries were considered by Nash and Ludäscher in [Nash and Ludäscher 2004a; 2004b]. Florescu et al. [Florescu et al. 1999] look at integrating access restrictions into a cost-based optimizer.

In the absence of integrity constraints, querying with access patterns amounts to a limitation on the search space, restricting the ordering of atoms within a query plan. In contrast, schemas with integrity constraints and access patterns can simultaneously restrict the search space (via access restrictions) and extend it (via integrity constraints, which allow relations outside of the query to become relevant).

Comparison to work of Nash, Segoufin, and Vianu. The departure point for this paper is the work of Segoufin and Vianu [Segoufin and Vianu 2005] and Nash, Segoufin, and Vianu [Nash et al. 2010], which introduce the idea of relating a semantic property of a query relative to an interface restriction — in their case, determinacy of a query by views —- with the existence of a reformulation of the query over the views.

This idea is not phrased in terms of algorithms going from proofs of the semantic property to reformulations, since [Nash et al. 2010] does not work by default in the setting of unrestricted instances (as we do), but in the setting of finite instances, and over finite instances semantic entailment is not necessarily the same as the existence of a proof. Indeed, [Nash et al. 2010] investigates the question of determinacy vs. reformulation in the finite in detail.

[Nash et al. 2010] discovered that even in the setting of conjunctive query views, conjunctive queries may have a relational algebra reformulation but no conjunctive query reformulation. They defined the corresponding semantic property for positive existential reformulation, the analog of our access-monotonic-determinacy definition in the view case. However, there is no investigation of the different notions of determinacy that correspond to different kinds of target queries in [Nash et al. 2010].

We give theorems not for reformulation as a query but for plan-generation using access methods. We emphasize the semantic property and proof goal that corresponds to each notion of plan, stressing that one can go effectively from the proof to the plan. Although this correspondence relies on considering equivalence over all instances, we show that it can be "pushed down" to equivalence over finite instances for many classes of constraints considered in practice (e.g. those with terminating chase).

In short our work can be seen as generalizations of [Nash et al. 2010] to the setting with access patterns and constraints, stressing the relationship between semantic properties, proofs, and plans.

Our algorithm for generating RA-plans is extremely close to the construction in the case of reformulation over conjunctive query view definitions by Nash, Segoufin, and Vianu on page 21:29 of [Nash et al. 2010]. The construction of [Nash et al. 2010] is quite specific to the conjunctive query view case.

Comparison with the Chase and BackChase. The Chase and Backchase (C&B) is a common technique for reformulating a conjunctive query Q. It originated in papers of Popa [Popa 2000], Tannen, and Deutsch (e.g., [Deutsch et al. 2006; 1999]). The technique exists in various forms, but a frequently cited version of it matches our SPJ-plan algorithm in the special case of "vocabulary-based access restrictions": we have a distinguished set of predicates $\mathcal{T}$ which the reformulated query should be over, and we desire a conjunctive query reformulation $Q_{\mathcal{T}}$. In terms of constraints, there are versions of the C&B for constraint sets including both tuple-generating dependencies, but also equality-generating dependencies (EGDs). The main assumption is that we have a notion of the chase which terminates. Such a variant is well-known for TGDs and EGDs, but the requirements for termination of the chase become very strong [Onet 2013]. Below we will explain the connection between the C&B and the results in this paper only for TGDs.

The idea of the C&B is to first produce a "universal plan" by chasing the canonical instance of query Q with the constraints to get a collection of facts U: this is the "chasing phase". Then we search for a smaller plan that uses only the distinguished predicates from $\mathcal{T}$. We do this by selecting a set S of such facts within the chase, re-chasing it by tracing out the closure of S in U under the dependencies. If this closure has a match of the query Q (that is, a homomorphism mapping the free variables of Q to the corresponding constant) then we know that the set of atoms S, when converted to a query Q_S, is in fact equivalent to Q under the constraints. This is the "back-chasing" phase. Ideally, it will select a set S that is minimal with respect to having a chase-closure with a match for Q, thus producing a Q_S that is a minimal reformulation [Ileana et al. 2014]. By maintaining auxiliary information about the way in which the atoms in U are generated in the chase, the back-chasing phase can be made more efficient. For example, in Ileana et al. [Ileana et al. 2014], provenance information is maintained to speed up the back-chasing phase.

We have looked not at generating a query, but at plan-generation in the presence of access restrictions and constraints, and the technique given here — auxiliary schemas, two copies of the relation symbols, etc. —- may seem very distinct from the C&B. But we will explain here that in the case where the methods overlap — that is, where one is interested in generating an SPJ query over a subset of the relations, for constraints having a terminating chase — our approach is a variation of the C&B.

Recall that in Algorithm 2 we first form an initial chase using the Sch constraints Σ, and then apply steps consisting of firing an accessibility axiom followed by "follow-up rules" – the application of copies of the constraints. Instead of firing an accessibility axiom to explicitly generate a fact of the form $\mathsf{InfAcc}R(\vec{c})$, we could simply decorate the fact $R(\vec{c})$ by a special predicate B (for "back-chased") and then propagate predicate B through the initial chase. Thus choosing a path of accessibility axioms corresponds exactly to choosing a sequence of distinct atoms $s_1 \dots s_n$. The result of our plan-generation algorithm is a physical plan implementing the conjunctive query corresponding to the underlying set of atoms $\{s_1 \dots s_n\}$. If we take as a cost metric the number of access commands then our cost-based method will automatically minimize this among proof-based plans. Proposition 6.2 argues that the resulting plan will minimize the number of atoms in the corresponding query. Thus in particular, this will allow us to produce a minimal reformulation.

In summary, from a high-level the C&B method corresponds to our approach as follows: the chasing phase corresponds to applying the Sch constraints, a choice of atoms corresponds to firing accessibility atoms, while the back-chasing phase can be seen as applying the $\mathsf{InfAccCopy}$ constraints.

There are some distinctions between Algorithm 2 and the C&B:

— Algorithm 2 explores different chase *sequences*, while the C&B takes an unordered view of the chase, producing a set of atoms which are turned into a query. For binding pattern-based access-restrictions, the ordering of the chase corresponds to the ordering of accesses, which is critical to making the plan executable. For "vocabulary-based restrictions", where a subset of the relations are made accessible (as in the setting of views), Algorithm 2 can explore plans corresponding to different join orders, as discussed in Example 6.7.

— Algorithm 2 is cost-based. While the classic C&B algorithm deals only with getting minimal reformulations, this is not a fundamental limitation of the C&B approach, and cost-based extensions are considered in Popa's thesis [Popa 2000]

— By considering the back-chasing phase as a re-tracing of the original chase graph, the C&B can speed up the process of checking equivalence of S. This viewpoint is critical to the optimizations performed in C&B papers, such as Meier's [Meier 2014] and Ileana et al.'s [Ileana et al. 2014]. This optimization is usually presented in the setting of terminating chase, although it could be modified in the same we as we do to the setting of broader classes (e.g. GTGDs), creating the chase up to a point sufficiently large that a proof will be found if there is one. Although C&B usually couples a "proof-to-reformulation" algorithm with a particular algorithmic strategy of forming the full universal chase first, these two could also be decoupled, as our algorithms are presented here. For instance, one could interleave chasing the canonical database and "back-chasing" (choosing some atoms and seeing their consequences), and this might have advantages when the full chase is large or infinite.

The usual presentation of the C&B, as well as Algorithm 2, are focused on CQ reformulations/SPJ-plans. The C&B has been extended to deal with unions of conjunctive queries with atomic negation (see the discussion of [Deutsch et al. 2007] in this section), but not to relational algebra reformulation. In this work we have considered existential plan-generation, but have shown that it collapses to the SPJ-plan-generation when inputs are constraints are TGDs. We have also considered relational algebra plan-generation.

In considering the entailments and semantic properties corresponding to different kinds of plans, we have underscored the connection between semantic properties like determinacy, proofs of their entailments, and plan-generation. This connection is the main contribution of our paper. The conference paper [Benedikt et al. 2014] extends this approach presented here to first-order constraints, while the later work [Benedikt et al. 2015] discusses implementation and experiments. We defer a comparison of the C&B with these later developments to another work.

Comparison with [Deutsch et al. 2007]. The first paper on querying with integrity constraints and access patterns is Deutsch, Ludäscher, and Nash [Deutsch et al. 2007].

[Deutsch et al. 2007] does not define a plan language, but rather deals with getting an "executable UCQ": a UCQ query that can be executed using the access patterns in the obvious way. We have shown in the electronic appendix that these correspond to our notion of $USPJ^{\neg}$-plan in expressiveness. Our Theorem 4.6 shows that if one starts with a CQ and the constraints consist only of TGDs, negation and union are not necessary to answer the query. But [Deutsch et al. 2007] allow the source query Q to be a $USPJ^{\neg}$ query, and allow constraints with disjunction and atomic negation on both sides. Thus our result does not apply to their setting. Although the constraints and source queries considered in [Deutsch et al. 2007] are richer than those dealt with here, the algorithms are specific to the case where the chase terminates.

[Deutsch et al. 2007] deal with the *existence problem* of determining whether a query has an equivalent $USPJ^{\neg}$ executable query, as well as the problem of finding such a query if it exists.

In Section 4 of the paper they define an algorithm for the existence problem:

(1) Apply the chase procedure to the original query Q with the constraints until termination. [Deutsch et al. 2007] chase the queries directly to get another query, rather than dealing with the corresponding canonical database to get an instance. Thus for them the result of the chase is another query Q_1.
(2) Form a query Q_2 as the "answerable part" of Q_1: this is (informally) a maximal subquery of Q_1 that can be generated using the access patterns.
(3) Chase Q_2 to get a new query Q_3, and check whether Q_3 is contained in the original query Q.

In the case of TGDs with terminating chase, this procedure matches our approach for SPJ-plans. The first step, chasing Q, corresponds in our setting to generating consequences using chase steps for the canonical database of Q_0 with the original copy of the constraints. The second step corresponds to applying our "forward accessibility axioms", or equivalently to taking the accessible part of the instance generated in the first step. The final step corresponds to applying the InfAccCopy constraints and checking for a match of the copy of Q.

Note that [Deutsch et al. 2007] utilize the algorithm above in the setting of their more general constraints, by defining a variant of the chase with disjunction and negation, and a notion of "answerable part" of a query that applies to $USPJ^{\neg}$ queries. This extension is not subsumed by the approach in this paper, since we do not deal with disjunction and negation in constraints. However, it is closely related to the approach for $USPJ^{\neg}$-plans in [Benedikt et al. 2014]. The extended definition of "answerable part" to handle atomic negation corresponds to applying both the forward accessibility axioms and a restriction of the backward accessibility axioms, with the restriction being that all variables x_i in the atom $R(x_1 \ldots x_n)$ must satisfy accessible. This variation of the accessible schema and its relationship to $USPJ^{\neg}$-plans is investigated in [Benedikt et al. 2014].

In Section 7 of [Deutsch et al. 2007], the authors provide another algorithm for the problem, which proceeds by augmenting the constraints with a new set of auxiliary axioms capturing accessibility (denoted Σ_D), and a derived query (denoted there as $dext(Q)$). The main result says that for some classes of constraints (those with "stratified witnesses") a source query Q has an executable $USPJ^{\neg}$ rewriting iff Q is contained in $dext(Q)$ with respect to the enhanced schema. Again, in the case of TGDs with terminating chase, this coincides with our technique. [Deutsch et al. 2007] emphasize the second algorithm as a way of reducing rewriting with access patterns and constraints to rewriting under constraints alone.

First-order/relational algebra rewritings, and their distinction from positive existential or SPJ rewritings, are not covered in [Deutsch et al. 2007]. Neither do they consider the relationship of rewritability to semantic properties. However, the semantics-to-syntax approach we take here is related to results and discussions in Section 9 of [Deutsch et al. 2007]. In Theorem 22, they prove that their notion of executable query covers all $USPJ^{\neg}$ queries that could be implemented using the access methods. Although the theorem is phrased using a Turing Machine model, the proof shows that every access-determined $USPJ^{\neg}$ query has a rewriting that is executable in their sense. We make use of this result in our analysis of expressiveness in Section 3.

Relationship to [Bárány et al. 2013]. The work of Bárány et al. [Bárány et al. 2013] deals also with integrity constraints and access patterns. Instead of plans, as here, it targets a relational algebra query that runs over the accessible part. Thus executing a rewriting will always require exhaustively generating the accesses. It defines the notion of access-determinacy used in this paper and obtains tight bounds on the complexity of detecting access-determinacy for constraints in guarded logics (e.g. the guarded negation fragment, inclusion dependencies). It also shows that access-determined conjunctive queries over guarded constraints always have rewritings that are in the guarded negation fragment. The distinction between relation algebra, existential, and positive existential rewriting is not studied in [Bárány et al. 2013]. Indeed, it is incorrectly claimed in [Bárány et al. 2013] that the notions coincide for GTGD constraints.

Comparison to [Benedikt et al. 2014; Benedikt et al. 2014; 2015]. This paper is an extension of the extended abstract [Benedikt et al. 2014]. That paper introduced a method for generating plans from proofs in the presence of general first-order constraints, with the general technique being based on interpolation, which we explain below. This paper deals with a special case of the method, which was emphasized in [Benedikt et al. 2014], in the case of constraints given by TGDs. Full proofs were not included in [Benedikt et al. 2014]. The scope of several theorems has been enlarged (e.g. covering non-boolean queries), while several new results — Theorem 4.6, concerning the collapse of $USPJ^{\neg}$-plans to SPJ-plans for TGDs, and Proposition 6.2 concerning dominance with respect to executable queries — have been added in this paper.

[Benedikt et al. 2014] briefly discussed heuristic optimization for search. In later work, [Benedikt et al. 2015] , the authors explored methods for making the generation of proofs from plans more efficient. The demonstration paper [Benedikt et al. 2014] applies the ideas in this paper to querying over web services.

Chase-based plan-generation and Interpolation-based plan-generation. The Craig Interpolation theorem [Craig 1957] states that whenever we have first-order formulas φ_1, φ_2 and φ_1 entails φ_2, there is a formula φ such that

— φ_1 entails φ, φ entails φ_2
— φ uses only relation symbols occurring in both φ_1 and φ_2, and only constants occurring in both φ_1 and φ_2

Such a φ is said to be an *interpolant* for the entailment of φ_2 by φ_1.

Craig showed that interpolation could be used to transform proofs of a certain semantic property of a query into a syntactic representation that enforces that property. Craig did this for a property called "implicit-definability", which is related to the notion of determinacy used in the later work of Segoufin and Vianu [Segoufin and Vianu 2005]. This idea of using interpolation to go from "semantics to syntax" has since been applied to a number of semantic properties by logicians (e.g. [Otto 2000]), but without addressing algorithmic concerns.

We explain briefly how interpolation can be used to generalize the proofs-to-plan approach we provide here. Consider the inductive algorithm given to create SPJ-plans from chase proofs, Algorithm 1. If we translate the plans to formulas, we see that the output of the algorithm is an interpolant for the entailment

$$Q \wedge \Sigma \models (\mathsf{Ax}_{\mathsf{For}} \wedge \Sigma' \rightarrow \mathsf{InfAcc}Q),$$

where Σ is a conjunction of the Sch constraints, Σ' a conjunction of the $\mathsf{InfAccCopy}$ constraints, and $\mathsf{Ax}_{\mathsf{For}}$ is a conjunction of the forward accessibility axioms. Consider the case of "vocabulary-based restrictions", where every relation either has no access methods or is an input-free access methods. The common non-schema constants are exactly the ones corresponding to free variables, and the common relations are those that have an access method. Therefore an interpolant will be a formula using only the relations that have an access method. Thus an interpolation algorithm can allow us to get reformulations in the vocabulary-based setting. The RA algorithm can similarly be seen as computing an interpolant for an entailment involving both forward and backward accessibility axioms, and for vocabulary-based restrictions such interpolants correspond to first-order reformulations over the relations that have an access.

The relationship between plan-generation and interpolants suggest that an alternative approach to our results would be to prove an appropriate interpolation theorem and argue that for any access schema the interpolants could be converted into plans. The advantage of an interpolation-based approach is that it could be applied to arbitrary first-order constraints, where chase proofs are not available. We do not have space to explain this in detail here, but such an approach is possible. One requires a proof system that generalizes the chase to arbitrary first-order constraints, and tableau proofs provide such a system. One also requires a strengthening of Craig's interpolation theorem guaranteeing that the interpolant can be converted to a plan making use of the access methods. In the conference paper [Benedikt et al. 2014] such a theorem is stated and its proof is sketched. Using this interpolation theorem, [Benedikt et al. 2014] derives theorems relating semantic properties, proofs, and plans, some of them generalizing the ones given here.

Comparison with [Toman and Weddell 2011]. Chapter 5 of the book of Toman and Weddell [Toman and Weddell 2011] outlines an approach to reformulating queries with respect to a physical schema that is based on proofs. They discuss proofs using the chase algorithm, as well as an extended proof system connected to Craig Interpolation, remarking that the latter can synthesize plans that are not conjunctive. Access methods are not the focus of the work, but [Toman and Weddell 2011] outline an approach for handling them heuristically, by post-processing formulas so that they can be executed using the access methods.

Other related work. Several other works provide algorithms for querying in the presence of both access patterns and integrity constraints. Duschka et al. [Duschka et al. 2000] include access patterns in their Datalog-based approach to data integration. They observe, following [Li 2003], that the accessible data can be "axiomatized" using recursive rules. We will make use of this axiomatization (see the "accessibility axioms" defined later on) but establish a tighter relationship between proofs that use these axioms and query plans.

REFERENCES

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison-Wesley.

AFRATI, F. N. 2011. Determinacy and query rewriting for conjunctive queries and views. *Theoretical Computer Science 412,* 11, 1005–1021.

BÁRÁNY, V., BENEDIKT, M., AND BOURHIS, P. 2013. Access restrictions and integrity constraints revisited. In *ICDT*.

BÁRÁNY, V., GOTTLOB, G., AND OTTO, M. 2010. Querying the guarded fragment. In *LICS*.

BÁRÁNY, V., TEN CATE, B., AND SEGOUFIN, L. 2011. Guarded negation. In *ICALP*.

BENEDIKT, M., LEBLAY, J., AND TSAMOURA, E. 2014. Proof-driven query answering over web-based data. In *VLDB*.

BENEDIKT, M., LEBLAY, J., AND TSAMOURA, E. 2015. Querying with access patterns and integrity constraints. In *VLDB*.

BENEDIKT, M., TEN CATE, B., AND TSAMOURA, E. 2014. Generating low-cost plans from proofs. In *PODS*.

CRAIG, W. 1957. Linear reasoning. a new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic 22,* 03, 250–268.

DEUTSCH, A., LUDÄSCHER, B., AND NASH, A. 2007. Rewriting queries using views with access patterns under integrity constraints. *Theoretical Computer Science 371,* 3, 200–226.

DEUTSCH, A., NASH, A., AND REMMEL, J. 2008. The chase revisited. In *PODS*.

DEUTSCH, A., POPA, L., AND TANNEN, V. 1999. Physical data independence, constraints, and optimization with universal plans. In *VLDB*.

DEUTSCH, A., POPA, L., AND TANNEN, V. 2006. Query reformulation with constraints. *SIGMOD Record 35,* 1, 65–73.

DUSCHKA, O., GENESERETH, M., AND LEVY, A. 2000. Recursive query plans for data integration. *The Journal of Logic Programming 43,* 1, 49 – 73.

FAGIN, R., KOLAITIS, P. G., MILLER, R. J., AND POPA, L. 2005. Data exchange: Semantics and query answering. *Theoretical Computer Science 336,* 1, 89–124.

FLORESCU, D., LEVY, A. Y., MANOLESCU, I., AND SUCIU, D. 1999. Query optimization in the presence of limited access patterns. In *SIGMOD*.

ILEANA, I., CAUTIS, B., DEUTSCH, A., AND KATSIS, Y. 2014. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*.

LI, C. 2003. Computing complete answers to queries in the presence of limited access patterns. *VLDB Journal 12,* 3, 211–227.

LI, C. AND CHANG, E. 2000. Query planning with limited source capabilities. In *ICDE*.

LI, C. AND CHANG, E. 2001. Answering queries with useful bindings. *ACM Transactions on Database Systems 26,* 3, 313–343.

MAIER, D., MENDELZON, A. O., AND SAGIV, Y. 1979. Testing implications of data dependencies. *ACM Trans. Database Syst. 4,* 4, 455–469.

MEIER, M. 2014. The backchase revisited. *VLDB Journal 23,* 3, 495–516.

NASH, A. AND LUDÄSCHER, B. 2004a. Processing first-order queries under limited access patterns. In *PODS*.

NASH, A. AND LUDÄSCHER, B. 2004b. Processing union of conjunctive queries with negation under limited access patterns. In *EDBT*.

NASH, A., SEGOUFIN, L., AND VIANU, V. 2010. Views and queries: Determinacy and rewriting. *ACM Transactions on Database Systems 35,* 3.

ONET, A. 2013. The chase procedure and its applications in data exchange. In *Data Exchange Intregation and Streams*.

OTTO, M. 2000. An interpolation theorem. *Bulletin of Symbolic Logic 6,* 4, 447–462.

POPA, L. 2000. Object/relational query optimization with chase and backchase. Ph.D. thesis, U. Penn.

RAJARAMAN, A., SAGIV, Y., AND ULLMAN, J. D. 1995. Answering queries using templates with binding patterns. In *PODS*.

SEGOUFIN, L. AND VIANU, V. 2005. Views and queries: determinacy and rewriting. In *PODS*.

TOMAN, D. AND WEDDELL, G. 2011. *Fundamentals of Physical Design and Query Compilation*. Morgan Claypool.

ULLMAN, J. D. 1989. *Principles of Database and Knowledge-Base Systems, V2*. Comp. Sci. Press.