Querying with Access Patterns and Integrity Constraints*

Michael Benedikt, Julien Leblay, and Efthymia Tsamoura
Oxford University, UK
name.surname@cs.ox.ac.uk

ABSTRACT

Traditional query processing involves a search for plans formed by applying algebraic operators on top of primitives representing access to relations in the input query. But many querying scenarios involve two interacting issues that complicate the search. On the one hand, the search space may be limited by access restrictions associated with the interfaces to datasources, which require certain parameters to be given as inputs. On the other hand, the search space may be extended through the presence of integrity constraints that relate sources to each other, allowing for plans that do not match the structure of the user query.

In this paper we present the first optimization approach that attacks both these difficulties within a single framework, presenting a system in which classical cost-based join optimization is extended to support both access-restrictions and constraints. Instead of iteratively exploring subqueries of the input query, our optimizer explores a space of proofs that witness the answering of the query, where each proof has a direct correspondence with a query plan.

1. INTRODUCTION

Query processing involves generating plans for accessing a given set of datasources, with plans typically given in some variant of relational algebra. However many applications, such as data integration, impose both restrictions and expansions of the search space. Interfaces to datasources, particularly in the Web data integration setting, often come with access restrictions, requiring certain parameters to be given as inputs, thus restricting the space of plans. On the other hand, integrity constraints between sources can radically expand the search space, allowing for plans that bear little resemblance to the source query. These two aspects have a rich interaction. Constraints may relate two sources that have very different access – for example, one that has unrestricted access and one that has no access at all, such as a virtual table used in data integration. Thus the presence of both integrity constraints and access restrictions may make it both *possible* and *necessary* to consider plans using relations that are not mentioned in the user query.

*Supported by EPSRC grant EP/H017690/1, Query-driven Data Acquisition from Web-based Data Sources

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 6
Copyright 2015 VLDB Endowment 2150-8097/15/02.

In this paper we give an approach to query optimization considering both access restrictions, integrity constraints, and their interaction. Our approach explores the space of plans for implementing a query by exploring a space of proofs that witness the answering of the query, where each proof has a direct correspondence with a query plan. Proof steps can represent derivation of implicit information using constraints. Proofs can also capture reasoning about access restrictions on sources. Consider, for example, the reasoning that a parameter needed for one data access (e.g. an argument to a web service) can be provided as output from another data access. In our framework, this will correspond to matching the hypotheses of one proof (the fact that a parameter is assumed to be available) with one of the conclusions of another proof (the fact that a certain parameter is returned). While classical query optimization involves searching for good ways of composing expressions (e.g. join ordering) to answer a query, in our system optimization involves searching for good ways of constructing proofs from subproofs.

EXAMPLE 1. *Consider a schema that exposes the relations* $\mathsf{Profinfo}(\mathsf{Pname}, \mathsf{Profid})$, $\mathsf{OfficeIn}(\mathsf{Profid}, \mathsf{Offid})$ *and* $\mathsf{Offices}(\mathsf{Offid}, \mathsf{Bname})$. *Informally,* Profinfo *contains names and ids of professors,* OfficeIn *tells which office a professor has, while* Offices *maps office ids to building names.*

The following query Q asks for the names of professors in Van Vleck hall:

$$\{\mathsf{Pname} \mid \exists \mathsf{Profid}\ \mathsf{Offid}\ \mathsf{Profinfo}(\mathsf{Pname}, \mathsf{Profid}) \wedge \mathsf{OfficeIn}(\mathsf{Profid}, \mathsf{Offid}) \wedge \mathsf{Offices}(\mathsf{Offid}, \text{“Van Vleck”})\}$$

There is a service that allows access to Profinfo, *but it requires as input a professor's id.* OfficeIn *and* Offices *are GAV relations, i.e., virtual tables with no access at all. However,* OfficeIn *and* Offices *are related to a web-based source* $\mathsf{OfficeInfo}(\mathsf{Profid}, \mathsf{Bname})$ *that stores the building in which each professor's office is located, which allows access on the building name.*

Query Q can be answered by first doing a lookup in OfficeInfo *using input "Van Vleck" to get a list of* Profid*s, and then finding the names by doing a lookup on each id within* Profinfo. *But reasoning that this is the case requires making use of the integrity constraint that captures the informal semantics described above, i.e., the constraint asserting that* OfficeInfo *contains exactly the pairs* $(\mathsf{Profid}, \mathsf{Bname})$ *satisfying* $\exists \mathsf{Offid}\ \mathsf{OfficeIn}(\mathsf{Profid}, \mathsf{Offid}) \wedge \mathsf{Offices}(\mathsf{Offid}, \mathsf{Bname})$.

Accounting for the access methods is also critical in reasoning that this plan is correct. For example, we would not be able to answer a query asking for the office id *of professors in Van Vleck, since the ids are not accessible. Clearly the kind of plans we need will be quite different from the plans in traditional processing.* □

EXAMPLE 2. *Let the relations* Employee(eid, ename, ..., mgrid) *and* Manager(mgrid, deptmanaged, ...) *with* Employee *having two access methods, looking up on* eid *and* mgrid *respectively, and* Manager *having an access method with input* deptmanaged*. We have a referential constraint from* Employee *to* Manager *and also a referential constraint stating that every* mgrid *in* Manager *is the* eid *of an* Employee*. The query that asks for the names of employees in the mathematics department, as well as the names of their managers, is answerable: by first doing an access to* Manager*, selecting those* mgrid*s corresponding to mathematics, doing a lookup in* Employee *on* mgrid *to extract employee names, and finally doing a second lookup on* Employee*'s* eid *to extract the manager names. Notice that the integrity constraints here contain "cycles", since* Employee *points to* Manager *and vice versa. Many methods for reasoning with queries under constraints (those based on a "terminating chase" [11]) do not deal with such constraint classes.*□

We will start with an idea from earlier theoretical work [5], which takes as input a query Q and schema S with access restrictions and constraints, and generates a *proof goal* which corresponds to reasoning that the query can be answered using the integrity constraints but abiding by the access restrictions. The proof goal will have the property that each proof of the goal will correspond to a plan for answering the query. Thus by exploring these proofs, we can consider radically different query plans.

We will look at how this theory can translate into an effective query optimization algorithm, while both incorporating standard search heuristics for query optimization, such as the use of dynamic programming, and allowing the flexibility to produce plans that are not left-deep, since left-deep plans can lead to inefficiency in the presence of access restrictions (see Florescu et al. [10]). We will develop a compositional proof system for building up "proofs of answerability" in this work, along with an algorithm for converting these proofs to query plans. Using these two tools we will give a new recursive search procedure which begins with small proofs and combines them into larger proofs, until finally we find a proof of the entire proof goal, corresponding (via the proof-to-plan algorithm) to a plan that answers a query. Building up proofs from subproofs will be analogous to building up larger query plans from subplans in classical query optimization. We will show that this method can give both left-deep and "bushy" query plans, and that it can be adapted using the same variations used in classical query-optimizations – e.g. using either dynamic programming or heuristic search. On the other hand, we provide optimizations that are specific to the proof-driven context, chiefly by discarding proof states that are less interesting than those already discovered.

In summary, we present the first system for doing cost-based optimization focusing on access restrictions, constraints, and their interaction, along with an experimental evaluation of the performance of the system.

Related Work. This work concerns the interaction of access restrictions and constraints on query answering. The impact of access restrictions on query answering in the *absence* of constraints has been considered in a number of theoretical works (e.g. [13]). Deutsch et al. [7] was the first work to deal with query answering with constraints and access patterns from a theoretical perspective. They provide a decision procedure for determining whether a query has a feasible plan in the presence of access restrictions and constraints for which the chase terminates. Complexity bounds for this problem, as well as extensions to richer constraint classes, can be found in [2, 5]. In particular, [5] introduces the idea that plans for answering a query correspond to proofs that use two copies of the integrity constraints along with "accessibility axioms". We make heavy use of this correspondence here, but explore how it can be used within cost-based query optimization. The idea of reducing a query planning problem to verification that the query is "answerable" in turn has its origins in the work of Nash, Segoufin, and Vianu [15].

From the point of view of cost-based query optimization, the main prior work on integrating access restrictions into a rule-based optimizer is Florescu et al.'s [10], which is the departure point for this work. In the absence of constraints, access restrictions limit the possible join orderings that should be considered, since many are infeasible for execution. At the same time, they can lead to non-optimality of left-deep plans, thus requiring an expansion of the search space. Florescu et al. adapt traditional dynamic programming, searching equivalence classes of subqueries under an equivalence relation that is aware of access restrictions. We will introduce a notion of subquery equivalence (fact-equivalence, see Section 5) that can be seen as an extension of the one in [10] to the constraint-aware case. To deal with the expansion of the search space, [10] depart from traditional dynamic programming in employing a "best-first" search strategy, focusing the search on subplans that are optimal in a heuristic plan utility measure. We will use utility heuristics based on the constraint graph to achieve a similar reduction in the setting with integrity constraints (see Section 5).

There is also considerable amount of work on *querying with constraints, but in the absence of access restrictions*. The implementations of querying with constraints have focused on *query reformulation*, where the input query Q is pre-processed to form another query Q' that is equivalent to Q under constraints. The chase and backchase approach (C&B) originating in work of Deutsch, Popa, and Tannen [8, 17] is the most mature method for performing reformulation, focusing on the case of TGD constraints where the chase terminates. A number of optimizations of the chase and backchase have appeared recently [14, 11] that report drastic improvements in speed of reformulation.

While efficiency in the reformulation process has been investigated, the notion of cost used in optimizing reformulations is generally restricted to minimizing the number of query atoms: remaining cost considerations are delegated to a query engine. In C&B, physical query optimization, other than join ordering considerations, is accommodated to some extent by dividing up the schema into logical and physical subschema, with the latter corresponding to accesses. The output of reformulation is a query over the physical subschema, which is given a canonical physical implementation. But access ordering and cost considerations that depend upon ordering are not considered. *Note that in the presence of access restrictions, dealing with ordering issues is crucial to query optimization, since some orderings may not be realizable.*

The book of Toman and Weddell [18] overviews a broad approach to reformulation that maintains the logical/physical schema restriction, but adds access restrictions to the mix. A reformulation process that works for general first-order constraints is outlined, which produces a query over the physical schema, but one that may not abide by access restrictions. In a post-processing phase, the query is rewritten to obey access restrictions. The approach does not target integration with a database engine, and optimizing join ordering is not considered.

In contrast, our work does not target simple reformulation – e.g. producing a set of query atoms which need to be ordered and implemented by another optimizer. In addition, we deal with a broad class of cost functions, and we deal natively with ordering issues. We target a physical plan language that consists of a sequence of access and join operators. This requires us to revisit traditional join optimization issues in the light of both access restrictions (as

in [10]) and in the presence of constraints – we know of no other work that presents and evaluates a system that handles both issues.

Our general framework for constraint-based optimization does not depend upon termination of the chase, but applies to a wide range of constraints for which effective decision procedures exist. We thus see this work as a key step in making recent work on decision procedures for richer constraint languages (e.g. [6]) applicable to query optimization.

The methods presented in this paper were implemented in a system named PDQ, whose front-end for web-based data was presented as a demonstration in VLDB 2014 [4]. It can be downloaded from our project website and run as a stand-alone GUI[1].

2. DEFINITIONS

Our starting point will be a *schema* which describes a querying scenario, consisting 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 input 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.
- 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.
- A collection of *integrity constraints*, which we will always assume are given by sentences of first-order logic (interpreted under the active domain semantics [1]), using only relations and constants from the schema.

  We will give particular attention to 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 from the set C.

  A special subclass consists of *Guarded TGDs*, in which φ is of the form $R(\vec{x}) \wedge \varphi'$ where $R(\vec{x})$ contains all variables of φ'. These subsume *inclusion dependencies* (IDs): where φ is of the form $R(\vec{x})$ in which no variables are repeated and there are no constants, while ρ is also a single atom with no repeated variables or constants. IDs are also called "referential constraints".

Informally, the access methods give restrictions on how relations can be accessed. A standard example of relations with access methods comes from Web forms, where the input positions represent mandatory fields of the form.

We will use standard terminology for describing queries in first-order logic, including the notion of free variable, quantifiers, connectives, etc. [1]. A database instance (or just database) I for schema S assigns to every relation R in S a collection of tuples $I(R)$ of the right arity, in such a way that any integrity constraints of S are satisfied. 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.

We consider *conjunctive queries* (CQs), 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 and variables from $\vec{x}$ and $\vec{y}$ and/or constants from the schema as arguments. These are equivalent to queries built up from selection, projection, and join in relational algebra, and 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. Given a conjunctive query Q and instance I, $Q(I)$ is the result of evaluating Q on I. A *homomorphism* of CQ Q into some instance I is a mapping of the variables of Q to values of I that preserves all atoms of Q.

We say that a query Q *entails* another query Q' with respect to a set of integrity constraints if in any instance that satisfies the constraints $\forall \vec{x}\ (Q(\vec{x}) \rightarrow Q'(\vec{x}))$ holds. As with other notions, by default we deal here with arbitrary instances, not necessarily finite. Entailment of first-order logic formulas can be captured by proof systems, and thus it is equivalent to say that Q proves Q', for any complete proof system.

Chase proofs. In this work we will make use of the "forward-chaining" proof system known in the database literature as *the chase* [1, 16]. A proof in the chase can be rephrased as 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_i of each variable x_i 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*. Consider a set of facts I, a TGD $\delta = \forall \vec{x}\ \varphi(\vec{x}) \rightarrow \exists \vec{y}\ \rho(\vec{x}, \vec{y})$, and $\vec{e}$ such that $\varphi(\vec{e})$ holds but there is no $\vec{f}$ for which $\rho(\vec{e}, \vec{f})$ holds in I. A *chase step* for such a $\delta, \vec{e}$ adds facts to I that make $\rho(\vec{e}, \vec{f})$ true, where $f_1 \ldots f_k$ are new constants ("chase constants").

A *chase sequence* following a set of dependencies Σ consists of a sequence of instances $F_i : 1 \leq i \leq n$, where F_0 is the canonical database of Q and F_{i+1} is obtained from F_i by some firing of a dependency in Σ. A *successful chase sequence* witnessing that Q entails Q' is one that ends with an instance F_n such that Q' has a homomorphism that is *consistent with* Q: mapping the free variables of Q' to the constants corresponding to free variables of Q. We also say that there is a *match* for Q' in F_n.

We now have the following well-known result, stating that the chase is a complete proof system (e.g. [1])

PROPOSITION 1. *For any conjunctive queries Q and Q' (with the same free variables), and any TGD constraints Σ, Q entails Q' w.r.t. Σ iff there is a successful chase sequence following Σ witnessing this.*

Access plans and costs. We now describe the plan language that will be the target of our query optimizer, which will mix variants of relational algebra operators with abstractions for low-level access, abiding by the access methods in the schema. The language is still quite high-level, and admits a variety of implementations. The most natural one in the Web setting is to perform the algebra operators in middleware, while using the accesses to model bulk invocation of a web service or bulk extraction from a web form. We can also implement the plan language on top of a database manager (translating the language back to SQL), or mapping to the low-level access and operators in the database manager's plan language. To accommodate the set of plans that are produced from our proof-based approach, we will need a very flexible formalism for representing *open plans* – that is, plans that can not be executed stand-alone, but instead require an additional input relation, representing a set of values for plan parameters that may flow in from an external source.

A plan over a schema S with access methods is a term built up from typed operators. The basic types are the *relational types*. Plans with such types are standard "closed plans", requiring no input and returning a relation. These types are specified by giving the output attributes of the relation (here without domain datatypes, for simplicity). We denote such a type as $\{a_1 \ldots a_n\}$ where a_i are the

[1] http://www.cs.ox.ac.uk/projects/pdq/

output attributes. Types can then be built up via the function and product constructors.

We have an atomic operator for datasource access, $\mathsf{AccessOp}$, labelled with method mt on relation R and type $I \to O$, where I are the input attributes of mt and O are all attributes of R. We also allow $\mathsf{AccessOp}$ to be annotated with both a method mt and an additional mapping σ of a subset of the input positions of mt to values from the set C of schema constants: now the inputs I are the unmapped input attributes of mt. We also have atomic operators for relational algebra selections and projections.

Finally, we have generalized dependent join operators particularly appropriate for combining plans in middleware. For input attributes $a_1 \dots a_m$, $b_1 \dots b_j$ and output attributes $a_1 \dots a_n$ and $b_1 \dots b_k$, with $m \leq n, j \leq k$ we have a higher order operator $\overrightarrow{\bowtie}(P_1, P_2)$ operating on a plan P_1 with input attributes $a_1 \dots a_m$ and output attributes $a_1 \dots a_n$, and a plan P_2 with input attributes $b_1 \dots b_j$ and output attributes $b_1 \dots b_k$. The operator produces a plan with input attributes $\{a_1 \dots a_m\} \cup (\{b_1 \dots b_j\} - \{a_1 \dots a_n\})$ and output attributes $\{a_1 \dots a_n\} \cup \{b_1 \dots b_k\}$. A plan with no inputs is called *closed*; otherwise it is called *open*.

Terms are built from composing these operators, with the types of terms computed compositionally. The plans of interest to us will be the terms of either type $\{b_1 \dots b_n\}$ for attributes $b_1 \dots b_n$, or of type $\{a_1 \dots a_m\} \to \{b_1 \dots b_n\}$ for attributes $a_1 \dots a_m, b_1 \dots b_n$. That is, either a closed plan returning relations having attributes $b_1 \dots b_n$ or an open plan with input tuple having attributes $a_1 \dots a_m$. We can speak of the *input* and *output attributes* of such a plan. Every plan denotes a function mapping an input parameter matching the input attributes (empty if no input) to a set of tuples of the given output attributes.

The atomic plans generated for methods and for the relational operators are the obvious ones. Execution for $\overrightarrow{\bowtie}(P_1, P_2)$ proceeds, given an input parameter $\vec{t}$, by projecting $\vec{t}$ on the inputs for P_1, making a recursive call to the execution method to run P_1 on the projected tuple, joining the output back with $\vec{t}$, and then doing the same for P_2 on each resulting joined tuple. In the special case where P_1 has no input, this degenerates to a traditional dependent join, which can be optimized in many ways (e.g., discussion in [10]).

An example plan for answering the query in Example 1 would be $\overrightarrow{\bowtie}(\mathsf{AccessOp}(\mathsf{mt}_{\mathsf{OfficeInfo}}), \mathsf{AccessOp}(\mathsf{mt}_{\mathsf{Profinfo}}, \{\mathsf{Bname} =$ "Van Vleck"$\}))$. Here $\mathsf{mt}_{\mathsf{OfficeInfo}}$ is the access to table $\mathsf{OfficeInfo}$.

Given a schema S with access methods and constraints, a plan *answers* a query Q if for every instance I satisfying the constraints of S, 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.

For general constraints, query answering over all instances is not always the same as answering over finite instances. But for the constraints considered in this paper (TGDs with terminating chase and Guarded TGDs) the two notions of answerability are known to coincide [3].

Cost. A *plan cost function* associates every plan with a real-valued cost. The *minimal cost problem* for a schema with access restrictions and integrity constraints, query and cost function is the problem of finding a plan answering the query with minimal value of the cost function.

Our framework can work with a "black box" cost function on plans. If no information about the underlying sources is available, a default cost metric would associate each access method mt with a positive rational cost c_{mt}, and then the total cost of a plan whose access commands are calls to $\mathsf{mt}_1 \dots \mathsf{mt}_n$ (with possibly the same method repeated with different arguments) would be defined as $\Sigma_{i \leq n} c_{\mathsf{mt}_i}$. We refer to these as *simple cost functions* in the remainder, and we will provide refinements of the algorithms for this case.

3. REDUCING PLAN SEARCH TO PROOF SEARCH

We now review the idea of reducing plan search to proof search. The first step is to *axiomatize the access restrictions*, an idea that goes back to [9, 7]. Here we use a variation of the formalization of [5].

Given schema S_0, the *Accessible Schema for S_0*, denoted $\mathsf{AcSch}(S_0)$, is the schema without any access restrictions, such that:

- The constants are those of S_0.
- The relations are those of S_0, a copy of each relation R denoted $\mathsf{InfAcc}R$ (the *inferred accessible version of R*), a unary relation $\mathsf{accessible}(x)$ (*x is an accessible value*).
- The constraints of the accessible schema consist of
  - the constraints of S_0 (referred to as *original constraints* below),
  - *inferred accessible fact rules*, which consist of a copy of each of the original integrity constraints, with each relation R replaced by $\mathsf{InfAcc}R$.
  - *accessibility axioms*: for each access method mt on relation R of arity n with input positions $j_1 \dots j_m$ we have a rule:

$$\mathsf{accessible}(x_{j_1}) \wedge \dots \wedge \mathsf{accessible}(x_{j_m}) \wedge R(x_1 \dots x_n) \to \mathsf{InfAcc}R(x_1 \dots x_n) \wedge \bigwedge_i \mathsf{accessible}(x_i)$$

  In addition, we have $\mathsf{accessible}(c)$ for each constant c of S_0. Note that our assumption on the schema guarantees that two access methods can not generate the same accessibility axiom. Note that in presenting the accessibility axioms above, we drop the leading universal quantifiers for brevity.

Informally, the original copies of the relations represent data in the underlying "hidden" tables, while the inferred accessible copies represent the subset of the hidden data whose truth we could infer by first using accesses and then reasoning using the integrity constraints. The assertion $\mathsf{accessible}(c)$ indicates that the value c can be returned in some sequence of accesses. Thus the accessibility axioms represent the rules that allow one to move from a "hidden fact" (e.g., $R(c_1 \dots c_n)$) to an inferred accessible fact (e.g., $\mathsf{InfAcc}R(c_1 \dots c_n)$), corresponding to making an access. The axioms also capture that a value in a fact that is returned by an access is available for use in further accesses. The inferred accessible copy of the original schema represents the rules that allow one to take explicitly accessed information and infer new implicit facts via constraints: e.g., if from an accessibility axiom we have inferred $\mathsf{InfAcc}R(c_1 \dots c_n)$, and we have a referential constraint that says that R is contained in S, we would be able to infer $\mathsf{InfAcc}S(c_1 \dots c_n)$.

EXAMPLE 3. *In Example 1, there is an accessibility axiom capturing the semantics of the restricted access on* $\mathsf{OfficeInfo}$ *given by* $\mathsf{Ax}_1 : \mathsf{accessible}(\mathsf{Bname}) \wedge \mathsf{OfficeInfo}(\mathsf{Profid}, \mathsf{Bname}) \to \mathsf{InfAccOfficeInfo}(\mathsf{Profid}, \mathsf{Bname})$*, while the axiom* $\mathsf{Ax}_2 : \mathsf{accessible}(\mathsf{Profid}) \wedge \mathsf{Profinfo}(\mathsf{Name}, \mathsf{Profid}) \to \mathsf{InfAccProfinfo}(\mathsf{Name}, \mathsf{Profid})$*, corresponds to the restricted access on* $\mathsf{Profinfo}$. □

Given a query Q, its *inferred accessible version* $\mathsf{InfAcc}Q$ is obtained by replacing each relation R by $\mathsf{InfAcc}R$. Informally,

InfAcc$Q$ asserts that the truth of Q can be detected through making accesses and reasoning.

A main result of [5] is:

THEOREM 2. *Q entails* InfAcc$Q$ *with respect to the constraints in the accessible schema iff there is a plan that answers Q.*

Plans from chase proofs. We want to look at proofs witnessing that Q entails InfAcc$Q$ with respect to the generated schema axioms above. Since these axioms consists of TGDs, we can make use of the chase proof system, described in Section 2. Combining Theorem 2 and the completeness of the chase, Proposition 1, we immediately see that Q entails InfAcc$Q$ exactly when there is a chase sequence beginning with the canonical database of Q leading to a set of facts where there is a match for InfAcc$Q$.

EXAMPLE 4. *In Example 1, a proof must start with the canonical database of the query Q which consists of the facts* $\mathsf{Profinfo}(\mathsf{Name}_0, \mathsf{Profid}_0)$*,* $\mathsf{OfficeIn}(\mathsf{Profid}_0, \mathsf{Offid}_0)$ *and* $\mathsf{Offices}(\mathsf{Id}_0,$ *"Van Vleck"*$)$.

First, using the integrity constraint relating OfficeInfo *to the relations in the query, we infer* $\mathsf{OfficeInfo}(\mathsf{Profid}_0,$ *"Van Vleck"*$)$*. Using the accessible fact* accessible(*"Van Vleck"*) *and the axiom* Ax_1 *on* $\mathsf{OfficeInfo}(\mathsf{Profid}_0,$ *"Van Vleck"*$)$ *we derive the fact* $\mathsf{InfAccOfficeInfo}(\mathsf{Profid}_0,$ *"Van Vleck"*$)$*. Third, by applying the accessibility axiom* Ax_2 *to* $\mathsf{Profinfo}(\mathsf{Name}_0, \mathsf{Profid}_0)$ *we derive the fact* $\mathsf{InfAccProfinfo}\ (\mathsf{Name}_0, \mathsf{Profid}_0)$ *and, finally, using the copy of the integrity constraints, we add the facts* $\mathsf{InfAccOfficeIn}(\mathsf{Profid}_0, \mathsf{Offid}_1)$ *and* $\mathsf{InfAccOffices}(\mathsf{Offid}_1,$ *"Van Vleck"*$)$.

The accumulated facts now have a match for the query InfAcc$Q$ $=$ $\mathsf{InfAccProfinfo}(\mathsf{Name}_0, \mathsf{Profid}_0) \wedge \mathsf{InfAccOfficeIn}(\mathsf{Profid}_0, \mathsf{Offid}_0) \wedge \mathsf{InfAccOffices}(\mathsf{Id}_0,$ *"Van Vleck"*$)$.□

In searching for proofs, or even for proofs with the minimal number of firings of accessibility axioms, it is possible to restrict attention to *eager proofs*: those in which original constraints and inferred accessible fact rules always take preference over accessibility axioms. Formally, a proof is eager if it does not have a firing of an accessibility axiom at some step i, and then at a later step a rule firing involving the initial integrity constraints or their copies on the relations InfAcc$R$ that was already applicable at step i. It is clear that any proof can be turned into an eager proof by re-arranging the proof steps. The advantage of restricting to eager proofs is that their structure is simple: they consist of firing of the original integrity constraints on the canonical database, which we denote as an *initial chase*, followed by segments that consist of firing an accessibility axiom and then the firing of *follow-up rules*, the latter consisting of firing instances of the InfAcc copy of the integrity constraints.

Generating proofs compositionally. We will now provide a way of building up eager proofs compositionally in a "bottom-up" fashion, similar to the way a relational algebra plan for a query can be built bottom-up via subplans.

A *proof space* consists of (i) *initial chase facts* – informally, the facts generated by an initial chase of the canonical database of the query under the integrity constraints. The constants used in these are called the *initial chase constants*. (ii) A collection of *proof configurations*, or just *configurations* for short.

Proof configurations will be built up from atomic proofs via a composition operator, and each configuration is associated with (i) a collection of facts using initial chase constants called the *output facts OF*, which will always implicitly include the initial chase facts, (ii) a subset of the initial chase constants, called the *input chase constants IC*. IC will represent hypotheses that the proof uses about which values are accessible. We can derive from the output facts the collection of *output chase constants OC* of the configuration: those that are mentioned in the facts OF. A configuration with input constants IC and output facts OF represents a proof of OF using the rules of AcSch, starting from the hypothesis that each $c \in IC$ is accessible.

Configurations will be built in a "bottom-up" manner from basic configurations which correspond to firing of accessibility axioms.

Formally, configurations are built up inductively as follows:

- The basic configurations are $\mathsf{ApplyRule}(\mathsf{R}, \vec{b})$, where R is an accessibility axiom corresponding to method mt on relation R, and $\vec{b}$ is a binding of the universally quantified variables of R to chase constants or schema constants. The input constants are all those chase constants in $\vec{b}$ where the corresponding variable of R occurs within the R atoms of R at an input position of method mt. The outputs facts of the configuration are any inferred accessible facts produced be applying R with binding $\vec{b}$, as well as all facts that are consequences from these under the copy of the integrity constraints. Calculating these output facts requires a *consequence closure* procedure, discussed later.
- We say that an ordered pair of configurations $(\mathsf{conf}_1, \mathsf{conf}_2)$ is *non-trivial* if the output facts of conf_2 are not included in the output facts of conf_1 and vice versa, and if the ApplyRule subconfigurations of conf_1 and conf_2 do not overlap. Informally, non-trivial configurations represent proofs that produce incomparable conclusions, and without redundant rule firings. Whenever (x, y) is non-trivial, $\mathsf{Compose}(x, y)$ is a new configuration, which has input $I_1 \cup (I_2 - O_1)$ and output $O_1 \cup O_2$, and output facts all facts that are consequences of the union of the facts in x and y under the copy of the integrity constraints on the InfAcc relations. As above, calculating the set of facts requires the use of consequence closure.

A configuration is *successful* if its output facts contain a match for InfAcc$Q$. It is *hypothesis-free* if it has no input constants.

EXAMPLE 5. *In Example 1, our initial chase starts with the canonical database of the query and adds the facts* $\mathsf{OfficeInfo}(\mathsf{Profid}_0,$ *"Van Vleck"*$)$ *and* accessible(*"Van Vleck"*)*. One atomic proof configuration* conf_1 *is* $\mathsf{ApplyRule}(\mathsf{Ax}_1, \mathsf{OfficeInfo}(\mathsf{Profid}_0,$ *"Van Vleck"*$))$*, where* conf_1 *has empty input constants as* accessible(*"Van Vleck"*) *is part of the initial chase. The output facts of* conf_1 *include* $\mathsf{InfAccOfficeInfo}(\mathsf{Profid}_0,$ *"Van Vleck"*$)$*,* $\mathsf{accessible}(\mathsf{Profid}_0)$*, as well as the facts* $\mathsf{InfAccOfficeIn}(\mathsf{Profid}_0, \mathsf{Offid}_1)$ *and* $\mathsf{InfAccOffices}(\mathsf{Offid}_1,$ *"Van Vleck"*$)$*, since the output facts are closed under applying the copy of the intergrity constraints. Another configuration* $\mathsf{conf}_2 = \mathsf{ApplyRule}(\mathsf{Ax}_2, \mathsf{Profinfo}\ (\mathsf{Name}_0, \mathsf{Profid}_0))$ *has the input constant* Profid_0 *and a single output fact* $\mathsf{InfAccProfinfo}(\mathsf{Name}_0, \mathsf{Profid}_0)$.

Composing the first two proof configurations we get $\mathsf{conf}_3 = \mathsf{Compose}(\mathsf{conf}_1, \mathsf{conf}_2)$*, which has no input constants and whose output facts include the output facts of* conf_1 *unioned with those of* conf_2*. Configuration* conf_3 *has a match for the target query* InfAcc$Q$ *in its output facts, and thus* conf_3 *is an example of a successful and hypothesis-free configuration.*□

We will now define a compositional proof-to-plan algorithm ToPlan(conf) that takes any configuration conf and produces a (possibly open) plan, where the input parameters correspond to the input constants of conf. In particular, if conf is hypothesis-free, ToPlan(conf) will be a query over the original schema with no parameters.

The output relation of $\mathsf{ToPlan}(\mathsf{conf})$ will have attributes the output constants of conf (*output attributes* below). If conf is successful and hypothesis-free, $\mathsf{ToPlan}(\mathsf{conf})$ will, by convention, not return the whole set of outputs, but rather project onto the constants that correspond to free variables of the input query Q. Since $\mathsf{InfAcc}Q$ must have a match in such a configuration, these must necessarily be output attributes.

$\mathsf{ToPlan}(\mathsf{conf})$ is defined by induction on the structure of conf.

- We explain $\mathsf{ToPlan}(\mathsf{ApplyRule}(\mathsf{R},\vec{b}))$, for R an accessibility axiom and $\vec{b}$ a grounding of its quantified variables. Thus the grounding will be of the form:

  $$\mathsf{accessible}(b_{j_1}) \wedge \ldots \wedge \mathsf{accessible}(b_{j_m}) \wedge R(\vec{b}) \rightarrow \mathsf{InfAcc}R(\vec{b})$$

  where R has an access method mt with input $j_1 \ldots j_m$. We assume first for simplicity that $\vec{b}$ does not contain repeated values and that schema constants in $\vec{b}$ appear only in input positions of mt. We generate the plan $\mathsf{AccessOp}(\mathsf{mt},\sigma)$, with σ the mapping taking constants in $b_{j_1} \ldots b_{j_m}$ to the corresponding positions $j_1 \ldots j_m$. In the case that schema constants occur in output positions, we compose the operator above with selections filtering the output according to the schema constants. Repeated values in $\vec{b}$ result in composing with further equality selections.
- The plan produced by $\mathsf{ToPlan}(\mathsf{Compose}(\mathsf{conf}_1,\mathsf{conf}_2))$ will depend on the properties of $\mathsf{conf}_1,\mathsf{conf}_2$. If $\mathsf{conf}_1 \mathsf{conf}_2$ are hypothesis-free, then a traditional join method can be used on the common output constants of $\mathsf{conf}_1,\mathsf{conf}_2$. In the general case, we simply generate $\vec{\bowtie}(\mathsf{ToPlan}(\mathsf{conf}_1),\mathsf{ToPlan}(\mathsf{conf}_2))$.

Note that the collection of access commands in $\mathsf{ToPlan}(\mathsf{conf})$ will correspond exactly to the collection of accessibility axiom firings within conf. But the additional axioms are how we determine whether a configuration is successful, and thus whether the corresponding plan answers the query. To turn a successful hypothesis-free proof into a plan, we apply ToPlan as above and then add a top-level projection onto the attributes corresponding to the common free variables of Q and $\mathsf{InfAcc}Q$. As an optimization, this projection can be pushed deeper into the plan as a post-processing step.

EXAMPLE 6. *Returning to the configurations from Example 5, the atomic proof configuration* $\mathsf{conf}_1 = \mathsf{ApplyRule}(\mathsf{Ax}_1,\mathsf{OfficeInfo}(\mathsf{Profid}_0,$ *"Van Vleck"*$))$ *generates a plan* Plan_1 *that does a look-up in* OfficeInfo *on building name "Van Vleck", returning a set of* Profid*s. Configuration* $\mathsf{conf}_2 = \mathsf{ApplyRule}(\mathsf{Ax}_2,\mathsf{Profinfo}(\mathsf{Name}_0,\mathsf{Profid}_0))$ *generates plan* Plan_2 *taking as input a parameter tuple with one attribute* Profid_0*, performing look up on* Profinfo*, putting the resulting names and pids in an output table.*

The plan Plan_3 *for compound configuration* $\mathsf{conf}_3 = \mathsf{Compose}(\mathsf{conf}_1,\mathsf{conf}_2)$ *first does the lookup of* Plan_1*, then uses the results as inputs to* Plan_2 *via a dependent join on the resulting sets of* Profid*s. As pointed out in Example 5, there is a match of the query* $\mathsf{InfAcc}Q$ *to the output facts of* conf_3*, and thus* conf_3 *is a successful hypothesis-free configuration. We thus can get a successful plan by projecting the result on the attribute corresponding to the only free variables of the query, namely* Name_0. □

Consider an arbitrary chase proof witnessing that Q proves $\mathsf{InfAcc}Q$ using the accessible schema. Such a proof can be obtained as a simple kind of configuration, in which the right hand side of every Compose operation is an atomic ApplyRule configuration. We refer to these as *linear configurations*. Proof configurations generalize chase proofs to allow a more general way of composing proofs.

The following result, proved using the completeness of the chase procedure (e.g. [16]), shows the soundness and completeness of this method for generating plans:

PROPOSITION 3. *For any proof configuration* conf *for query* Q *and schema* S *with* conf *successful and hypothesis-free,* $\mathsf{ToPlan}(\mathsf{conf})$ *is an SPJ-plan that answers* Q *abiding by the schema. Conversely, if* Q *has any SPJ-plan , then it has one of the form* $\mathsf{ToPlan}(\mathsf{conf})$ *where* conf *is a linear configuration.*

In particular, we have completeness of linear proof configurations, which correspond to left-deep plans. Despite this, there are advantages to enabling the searching to range over a larger space of configurations, because it has been observed in Florescu et al. (Example 3.2 of [10]) that in the presence of access restrictions, restricting to left-deep plans can be particularly undesirable, forcing the creation of cross products.

Computing configurations and consequence closure. To construct a proof space we need to form the set of initial chase facts and we need to determine the output facts in each composition step. Both of these require a basic subprocedure of *consequence closure* – finding all consequences of some set of facts. Our high-level algorithms will be agnostic to how consequence closure is performed, and thus it applies to any logic that has an effective consequence closure operation. For TGDs where the chase necessarily terminates (for example, those that are weakly acyclic, or stratified [16]), we simply apply the rules of the chase algorithm until no rule is applicable. We also support Guarded TGDs, even when the chase does not terminate: we do this by cutting of the chase when it can be determined that further rule firings can not lead to a proof (see [6, 5]).

4. BASIC PROOF-DRIVEN SEARCH

We begin with a generic bottom-up algorithm, detailed in Algorithm 1, which searches for an optimal successful plan, building up a set of configurations Configs, while looking for one that is successful and for which the corresponding plan has low cost. We start by choosing a set of basic configurations of the form $\mathsf{ApplyRule}(\mathsf{R},B)$, corresponding to single-relation plans. We then extend Configs by adding new configurations built up using the binary operator Compose. The content of old configurations is never changed in this process.

This extension process proceeds in iterations. In the first phase of any iteration i we can add $\mathsf{Compose}(\mathsf{conf}_1,\mathsf{conf}_2)$ where $\mathsf{conf}_1,\mathsf{conf}_2$ are configurations from iteration $i-1$ such that neither of conf_i is both hypothesis-free and successful. After forming the new configuration $\mathsf{conf} = \mathsf{Compose}(\mathsf{conf}_1,\mathsf{conf}_2)$, we generate the corresponding plan and estimate its cost. We then check if it is unnecessary to add conf to the list of plan components – e.g. if we have already discovered that it is more expensive than a previously-discovered successful configuration. Assuming conf passes this test, we add it to the list of configurations, and we also check whether it represents a successful configuration. Both the formation of the initial facts and the implementation of the composition operator assume an effective consequence closure algorithm, as discussed earlier.

We can also restrict the shape of plans by considering only some subclass of *shape-restricted* configurations in each iteration of the loop. For example we can restrict to configurations that generate

Algorithm 1: Basic search

Input: query Q, schema S, height limit d
Output: plan BestPlan

1. Let PlanDag := $\emptyset$.
2. Initialize the set F_0 of facts obtained by firing original integrity constraint rules up to a termination condition.
3. Choose appropriate set of pairs $(\mathsf{R},\vec{b})$, where R is an accessibility axiom corresponding to access method mt and $\vec{b}$ is a binding that matches the input of mt, and add the corresponding configuration ApplyRule$(\mathsf{R},\vec{b})$ to PlanDag.
4. **if** *there is a successful hypothesis-free configuration in* PlanDag **then**
5. BestPlan := plan corresponding to the successful hypothesis-free configuration with lowest cost;
6. BestCost := *cost*(BestPlan)
7. **else**
8. BestPlan := $\perp$ BestCost$(n) = \infty$ BestCost := ∞ ;
9. $r := 1$;
10. **while** $r \leq d$ *and* PlanDag *has changed from previous* r **do**
11. **for** *all* $\mathsf{conf}_1,\mathsf{conf}_2$ *of height at most* r, $\mathsf{conf}_1,\mathsf{conf}_2 \in$ PlanDag *with* Compose$(\mathsf{conf}_1,\mathsf{conf}_2)$ *satisfying any additional shape-restriction* **do**
12. Let conf = Compose$(\mathsf{conf}_1,\mathsf{conf}_2)$;
13. **if** Cost(ToPlan((conf))) $\leq$ BestCost **then**
14. Add conf to PlanDag;
15. Determine if conf is successful by checking if InfAcc$Q$ holds;
16. **if** conf *is successful and hypothesis-free and* Cost(ToPlan(conf)) < BestCost **then**
17. BestCost = Cost(ToPlan(conf)) ;
18. BestPlan = ToPlan(conf);
19. $r := r + 1$;
20. return BestPlan;

only *left-deep plans*. Recall that these are the "linear configurations" – those where every composition has an ApplyRule configuration as a second argument.

Unlike a bottom-up join-ordering algorithm, the size of the original query does not bound the size of the output plan. We limit this by bounding the *height* of a configuration, where the height is the maximal nesting of binary operators within it. This bound will be a parameter to the algorithm.

In each round, the main loop chooses configurations to combine. If all pairs of configurations are considered, completeness is guaranteed. But as with traditional join processing, one can restrict attention to certain candidates (as discussed further in the paper).

Using the completeness of the chase, one can prove that if there is some plan for query Q, then Algorithm 1 will find it at some depth d. Further, for cost functions satisfying additional properties – e.g. the "simple cost functions" such as those that just count the number of access commands in a plan – one can show that *the algorithm will find a cost-optimal plan*.

Algorithm 2: Chase-friendly dynamic programming search

Input: query Q, schema S, height limit d
Output: plan BestPlan

1. Let PlanDag := $\emptyset$.
2. Initialize the set F_0 of facts obtained by firing original integrity constraint rules up to a termination condition.
3. Choose appropriate set of pairs $(\mathsf{R},\vec{b})$, where R is an accessibility axiom and $\vec{b}$ is a binding for rule R, and then generate the corresponding configurations ApplyRule$(\mathsf{R},\vec{b})$ adding them to PlanDag.
4. **if** *there is a successful hypothesis-free configuration in* PlanDag **then**
5. BestPlan := plan corresponding to the successful hypothesis-free configuration with lowest cost;
6. BestCost := Cost(BestPlan)
7. **else**
8. BestPlan := $\perp$ BestCost := ∞ ;
9. Reps := PlanDag ;
10. Set RepOf(conf) = (conf) and WitnessMap(conf) = identity, $\forall$ conf $\in$ PlanDag;
11. $r := 1$;
12. **while** $r \leq d$ *and* PlanDag *changed from previous* r **do**
13. CandConf := all conf $\in$ PlanDag such that dominated(conf) = SucDom(conf) = False;
14. Let CandPairs := all *shape-restricted* pairs $\mathsf{conf}_1,\mathsf{conf}_2$ with $\mathsf{conf}_1,\mathsf{conf}_2 \in$ CandConf not considered in previous rounds;
15. Let RepPairs := all pairs $\mathsf{conf}_1,\mathsf{conf}_2$ with $\mathsf{conf}_1,\mathsf{conf}_2 \in$ Reps such that $\exists\, \mathsf{conf}'_1,\mathsf{conf}'_2 \in$ CandPairs with RepOf$(\mathsf{conf}'_i) = \mathsf{conf}_i$ $i = 1,2$;
16. **for** $\mathsf{conf}_1,\mathsf{conf}_2 \in$ RepPairs *(in parallel)* **do**
17. Create conf = Compose$(\mathsf{conf}_1,\mathsf{conf}_2)$ via chasing and add to CandConf;
18. **for** $\mathsf{conf}_1,\mathsf{conf}_2 \in$ CandPairs $-$ RepPairs **do**
19. Let conf'_1 = RepOf(conf_1), h_1 = WitnessMap(conf_1), conf'_2 = RepOf(conf_2) , h_2 = WitnessMap(conf_2);
20. Retrieve conf$'$ = Compose$(\mathsf{conf}'_1,\mathsf{conf}'_2)$ from PlanDag;
21. Form conf = Compose$(\mathsf{conf}_1,\mathsf{conf}_2)$ by applying maps h_1, h_2 to conf$'$;
22. Add conf to PlanDag ;
23. **for** *all newly-added configurations* conf *from this round that are successful and hypothesis-free with* Cost(ToPlan(conf)) < BestCost **do**
24. BestCost = Cost(ToPlan(conf)); BestPlan = ToPlan(conf);
25. Update dominated, SucDom for all configurations;
26. Update WitnessMap, $\equiv_{\mathsf{Fact}}$, RepOf, Reps ;
27. $r := r + 1$;
28. return BestPlan;

5. DYNAMIC PROGRAMMING FOR PROOF-DRIVEN QUERYING

The basic template in the previous section proceeds in rounds, and is thus analogous to the building up of larger query plans from smaller in standard query processing. But in the standard theory a key optimization is not to keep all sub-plans in each round, but only the best within some class – e.g. the best plan that covers a given set of subrelations, or the best plan for a given "interesting order". We wish to apply the same idea in exploring proof configurations and their corresponding plans.

In the classical setting, we only compare subplans P_1 and P_2 that cover the same set of atoms, since two such plans have "the same

functionality" when used as components of a final plan. Thus in the classical case, we only compare components that appear in the same round of the search. In the proof-driven setting, two configurations may have the "same information" – producing the same of output facts using the same set of inputs – even though they appear in different rounds. Thus we will need an equivalence relation that compares configurations appearing in different rounds.

A related issue is that in traditional dynamic programming, if a subquery Q_1 covers strictly fewer atoms then another subquery Q_2, then Q_1 would be considered at a lower round of the recursion. Thus the "information ordering" on subqueries is compatible with the order of consideration in dynamic programming. In the presence of integrity constraints the situation is much different, because a query plan produces not only the explicit information returned by accesses, but also the implicit information implied by constraints. Thus a proof configuration C_1 produced at an early round – for example, one corresponding to a single access – may produce strictly more facts than a configuration produced at a later round, corresponding to multiple accesses. To deal with this, we will need to maintain not only a notion of "equivalent information" of components, but also a notion of "greater information". We will thus maintain a measure of information content of configurations, and keep only those which have maximal information content and minimal cost among those configurations with the same information content. This will be formalized via the notions of fact-domination, fact-equivalence, and domination, given below.

Comparing configurations. Our comparison of the information content of configurations must consider the input constants that they require and also the output facts that they produce – since clearly a configuration conf producing more output facts may be more valuable than one that produces less, even if conf' is more costly. The notion of having "no greater inputs and no fewer outputs" can be formalized via the appropriate notion of mapping. A mapping h from the chase constants of one configuration conf to the chase constants of another configuration conf' is *fact-preserving* if it preserves inferred accessible output facts in going from conf to conf', and if the h image of every input constant of conf is an input constant of conf'. If a mapping h is fact-preserving, then a successful plan p using conf can be converted into a successful plan $p' = h(p)$ for conf', by applying the mapping h to tables and commands in the obvious way. We write $\mathsf{conf} \preceq^h_{\mathsf{FactDom}} \mathsf{conf}'$ if h is a fact-preserving mapping from conf to conf', and $\mathsf{conf} \preceq_{\mathsf{FactDom}} \mathsf{conf}'$ (conf is fact-dominated by conf') if such an h exists. Say that configurations $\mathsf{conf}, \mathsf{conf}'$ are *fact-equivalent*, written $\equiv_{\mathsf{Fact}}$, if there is a bijective fact-preserving mapping h between them. Clearly, this is an equivalence relation.

Checking for isomorphism on the entire set of constants may be expensive. Thus in our implementation we strengthen the requirement further (over-approximating) by using "canonical names" (skolem terms) for constants generated via rules, and then requiring that the mapping preserve the name of all constants. This allows the $\preceq_{\mathsf{FactDom}}$ checks to reduce to a string match, and eliminates the choice of h (and hence the need to store it). Further, this will guarantee that any two fact-dominating mappings are compatible, which we will rely on below.

The above captures our notion of same/greater information. Let us now turn to cost.

Ideally, we would say that configuration conf is cost-dominated by configuration conf' (relative to a cost function) if for any successful superconfiguration conf^* of conf, substituting conf with conf' in conf^* yields a successful superconfiguration of conf' that is no more costly than conf^*. Clearly, in such a case we can discard conf. For simple cost-functions, this definition collapses to fact domination along with the easily-verifiable cost-related condition $\mathsf{Cost}(\mathsf{ToPlan}(\mathsf{conf})) \geq \mathsf{Cost}(\mathsf{ToPlan}(\mathsf{conf}'))$. For the moment we defer a discussion of how to estimate that configuration conf' is "hereditarily lower cost" than configuration conf in the sense above for more general cost functions, and simply assume this is captured by a relation $\mathsf{conf} \preceq_{\mathsf{CostDom}} \mathsf{conf}'$. We say that configuration conf is *dominated* by configuration conf', if $\mathsf{conf} \preceq_{\mathsf{FactDom}} \mathsf{conf}'$ and $\mathsf{conf} \preceq_{\mathsf{CostDom}} \mathsf{conf}'$.

We say that a configuration conf is *success-dominated* within a collection of configurations C if there is a successful configuration $\mathsf{conf}' \in C$ such that $\mathsf{conf} \preceq_{\mathsf{CostDom}} \mathsf{conf}'$. Assuming that the cost-domination relation is accurate, and that cost is monotone as access commands are added on to a plan, a success-dominated configuration should also not be explored. Neither assumption is universally true, but we make use of both as heuristics. We thus modify Algorithm 1 by calculating the $\preceq_{\mathsf{CostDom}}$ and $\preceq_{\mathsf{FactDom}}$ relations at every round, calculating the configuration that are dominated and success-dominated, keeping only those that do not fall into either category.

Reduction of chasing time. In classical dynamic programming the running time is proportional to the number of subqueries being considered, because the work done within the body of the recursion consists of operations that are not costly, such as computing the cost of a composed query from its components. In the proof-driven setting, forming a new configuration requires chasing, which can be extremely expensive. We will now consider ways to diminish chasing time.

The first optimization is to perform the chasing within a round in parallel. Instead of doing a simple iteration in forming new configurations from pairs $(\mathsf{conf}_1, \mathsf{conf}_2)$ as in Algorithm 1, we first perform a parallel loop over pairs, forming the output facts of each triple independently, by chasing the union of the output facts in conf_1 and conf_2.

The second optimization is to make use of the fact-equivalence relation. Notice that if $\mathsf{conf}_1 \equiv_{\mathsf{Fact}} \mathsf{conf}'_1$ via h, then for any compatible conf_2, if we have chased to form $\mathsf{Compose}(\mathsf{conf}_1, \mathsf{conf}_2)$, we can immediately obtain $\mathsf{Compose}(\mathsf{conf}'_1, \mathsf{conf}_2)$ by applying h, rather than having to re-chase.

Thus, assuming we maintain the $\equiv_{\mathsf{Fact}}$ relation, the witness maps, and a distinguished representative of each equivalence class for $\equiv_{\mathsf{Fact}}$, in the chasing step we can chase pairs that involve only two representatives, then apply the resulting maps to get the remaining composed configurations. To optimize further, we need not consider any candidate representative such that all equivalent elements are either cost- and fact-dominated or are success-dominated.

The optimized dynamic programming approach is shown in Algorithm 2. We omit the pseudo-code for domination and success-domination in the figure, assuming variable $\mathsf{dominated}$ in the algorithm is set to true exactly when a configuration is dominated, and similarly with variable SucDom. As with Algorithm 1, the algorithm is parameterized by a collection of shape-restricted configurations – e.g. linear configurations.

Cost estimation. We now return to the issue of the relation $\mathsf{conf}_1 \preceq_{\mathsf{CostDom}} \mathsf{conf}_2$. For hypothesis-free configurations we can apply ToPlan to both configurations and estimate their cost using a traditional cost function – e.g. via recursive estimation of output size and operation cost. In the case where one of $\mathsf{conf}_1, \mathsf{conf}_2$ is open, a conservative approach is to require conf_1 and conf_2 to have the same inputs while having an embedding of the plan operations of conf_1 into conf_2. We currently employ a more relaxed heuristic, summing up weights associated with each access, as in the simple cost case. Cost-domination can also be tuned to favor a

certain class of plans. For example, if we wish to be able to look at some "bushy" plans, but favor left-deep ones, we can simply exclude the possibility that a linear configuration is dominated by a non-left-deep plan. Excluding left-deep plans from domination by non-left-deep-plans will also ensure that the domination heuristic does not lose completeness of the search procedure.

Heuristics and shape-restrictions considered. Algorithms 1 and 2 are parameterized by the notion of shape-restrictions that one imposes on proofs/plans. Our implementation controls this via a "bushiness" parameter, defined inductively. Configurations of bushiness 0 are linear configurations, corresponding to left-deep plans. Configurations of bushiness $k+1$ allow only one top-level composition of configurations conf_1 and conf_2 of bushiness at most k, and further require that no subconfiguration of conf_1 or conf_2 is of form $\mathsf{Compose}(\mathsf{conf}'_1, \mathsf{conf}'_2)$ with conf'_1 of bushiness k and conf'_2 not an atomic configuration. We implement a search for bushiness k by iteratively running Algorithm 2 with shapes restricted to configurations, built up form composing with atomic $\mathsf{ApplyRule}$ configurations on the right, but starting with the results of the previous iteration: this is analogous to the iterative dynamic programming approach of Kossmann and Stocker [12].

Even with restrictions on bushiness, other heuristics on shape are needed to reduce the search space. Consider the case where every relation is accessible and where the constraints allow each atom R in the initial query to be inferred via n alternative means (e.g. n atomic access methods, or non-overlapping sequences of access methods). If k is the number of atoms in the query, then the search space is proportional to n^k, even when only left-deep plans are considered – we see this explosion in some of our sample queries (see Q11 in our benchmark discussed in Section 6). Thus as an additional heuristic, we associate each atom in the initial chase configuration its distance from the source query atoms in the "chase graph" which connects atoms A_1 and A_2 if a rule firing on A_1 generates A_2. Each configuration is then given a "query proximity rank" as the maximal distance of any chase atom exposed in the configuration. In line 14 of Algorithm 2, we then consider only the top d configurations conf'_2, ordered by distance, where d is a parameter given as an argument to the algorithm.

6. IMPLEMENTATION AND EVALUATION

We now describe the implementation of our planning system, denoted PDQ (Proof-Driven Querying), and perform an evaluation of it.

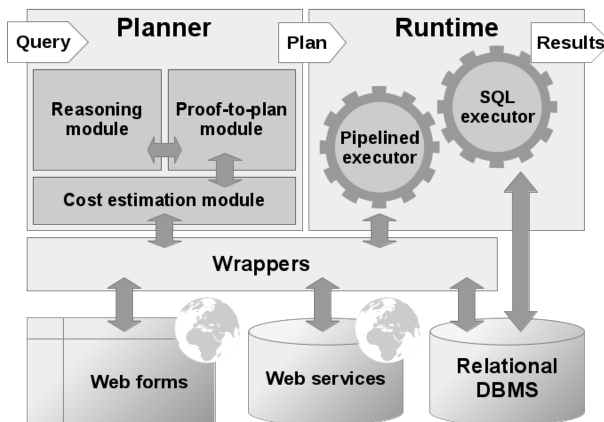

Figure 1: System's architecture.

The system's architecture is depicted in Figure 1. It features a planner, runtime, and wrappers.

Planner. A planner object implements the consequence closure operation in the algorithm. The implementation supported now is based on the chase algorithm: a chase-until-termination method is available for terminating chase classes, a blocking chase for constraints with non-terminating chase (cutting off the chase when it can be determined that further chasing can not lead to a proof, see [5]) for schemas containing guarded TGDs. Lastly, we support a bounded chase where we only fire up to a fixed number of times, passed as a parameter. The *proof-to-plan* module is in charge of building execution plans from proofs, making calls to the reasoner for consequence closure. Finally, a cost module to evaluate and compare the quality of our plans, which may interact with the underlying data management layer to collect statistics on the data. In particular, our cost module implements both (i) a simple cost function, as described in Section 2 and (ii) a typical "textbook" cost function, where the cost of a plan equals the estimated total number of I/Os performed by every operator. This function will be used in the experiments presented throughout the rest of this section, unless otherwise stated. Finally, (iii) the cost module can also delegate the cost computation to the underlying data management layer. Our latest release supports calls to external DBMS APIs when the schema relations are all stored on the same DBMS. Experiments using the PostgreSQL API will be presented on Section 6.5.

Runtime. PDQ has an execution environment with several execution modes. For example, in the iterator-type engine, a plan is a tree or DAG of physical operators consuming tuples from their children, and outputting tuples to their parents. Similar to the cost estimation module, the runtime can also delegate the execution to a third party RDBMS (typically when the cost was also delegated to that system).

Wrappers. The wrapper layer acts as an interface with the underlying storage in the case where execution is not delegated to the DBMS. Currently, two types of storages, relational databases and web services, are supported. The layer is responsible for reading and writing data, and also providing metadata to be used by the appropriate cost models (e.g. skew, connection latency, etc.) or by the planner (schema information, access methods given by indices or required input fields).

6.1 Experimental setting

Environment. We used the Amazon Cloud Computing environment to run our experiments, with each machine using 4 virtual CPUs (Intel Xeon / Ivy Bridge), with a maximum of 12GB of RAM allocated to the Java Virtual Machine. For TPCH-based experiments, the data resided in a PostgreSQL 9.2 database. Unless specified otherwise, we allocated 15 minutes for each planning session to complete.

Datasets. We evaluated PDQ on the TPCH benchmark on local, relational tables. Since our focus is on conjunctive queries, we handcrafted 13 CQs of different sizes (1 to 9 atoms) and shapes (including stars, chains, cycles, self-joins). Furthermore, we extracted the CQ components of seven of the TCPH standard benchmark queries (that is, eliminating uses of GROUP BY). The dependencies on the schema are those given by the foreign keys constraints. In addition, we created three arbitrary materialized views on the relations $\mathsf{region\text{-}nation}$ (V_1), $\mathsf{orders} - \mathsf{lineitem} - \mathsf{part} - \mathsf{customer}$ (V_2) and $\mathsf{orders} - \mathsf{lineitem} - \mathsf{part} - \mathsf{supplier}$ (V_3).

The TPCH benchmark does not come with any access restrictions. We therefore created manually various ranges of access patterns on top of the base schema. The schemas for which we report numbers in this paper allow free access to the V_2, V_3 and rela-

tions that are small or reference no other relation, namely region, supplier and part. Materialized views also feature access methods for each key and foreign key of the participating base tables. For other relations, our default schema features a "canonical" set of access restrictions guided by the foreign key constraints of the schema, i.e., every foreign attribute corresponds to a required input. We observed that for "natural" queries such as ours, where joins are on key/foreign key attributes, there are cross product-free left-deep plans. We also present results for an alternate schema, in which access restrictions are placed on attributes where we experimentally found this property was not met.

The database data has been generated using the 2.15.0 TPCH generator with scaling factor 0.001. In our experiments, the largest relation (line − item) contains 60K tuples. Note that the size of the data is immaterial to the performance of planning, but data statistics are used in evaluating the cost of the resulting plans.

6.2 Answerability

The first experiments investigate the benefits of constraint-awareness in the presence of access restrictions: how often our approach can find plans where a constraint-unaware optimizer will not find any, and how often it will find better plans than a constraint-unaware optimizer. We test this by not considering integrity constraints but only access patterns. In this experiment, we used the default schema (i.e., input attributes correspond to foreign key attributes). We report the numbers in Figure 2. The hatch-patterned bars indicate the cost of the best plans found when integrity constraints are disabled and we search the space of left-deep plan exhaustively. The plain bars correspond to cases where constraints were enabled. In this case, we employed a priority ordering heuristic, detailed in Section 6.3, to ensure completion within the time limit.

It is interesting to note that even with such minor restrictions, only six of our queries turn out to be answerable. Unanswerable queries correspond to missing bars on the figure. Moreover, the cost of plans for queries that were answerable was significantly reduced for half of these cases.

6.3 Planning time versus cost

We next evaluated the ability of PDQ to trade off planning time for cost, configuring the algorithm for linear configurations (hence left-deep plans) and using the ordering heuristic (see Section 5) of only exposing atoms the top d atoms in distance from the canonical database of Q in the chase graph. Results showing the impact as d varies are in Figures 3 and 4. We first ran the planner on all the queries for d=1, then 2, 3 and 5, where 5 is the length of the longest path in the chase graph. As in the previous experiment, we relied on the default schema.

Figure 3 shows planning time in milliseconds. For all but one query, planning time is a few seconds for d=1, then increases by 1 to 2 orders of magnitude as d is increased to 5. One outlier, Q_{11}, is our most demanding query. It has 8 atoms and triggers many constraints, leading to an initial chase with 22 atoms. For Q_{11}, planning with d=1 took 26 seconds, and increasing d to 2 caused the planner to time out (denoted by empty bars).

Figure 4 shows the cost of the best plan found for the same examples. Clearly, in most cases, the largest cost reduction is achieved when incrementing d from 1 to 2, with a gain of 2 orders of magnitude in half of the cases. Increasing d beyond 2 pays off mainly for queries whose relations are farther away from relations R for which there is no dependency with R in the right-hand side. This property can be checked easily at the beginning of the planning process. Overall, setting d to 2 by default yields a very good cost reduction with a minor penalty on planning time.

6.4 Impact of bushy plans

Recall that bushy plans may be necessary to avoid Cartesian products, and thus our implementation includes a "bushiness"' parameter, which controls how many nested bushy joins are considered in proof and plan search.

To test the impact of this parameter on plan cost, we use the alternate schema mentioned in Section 6.1, for which 15 of our queries do not have any Cartesian products-free left-deep plan. One query, Q_6, becomes unanswerable in this schema. Figure 5 shows a comparison of the cost of the best plans found for our set of queries under this scenario. In this figure, bushy corresponds to our own bushiness-friendly algorithm described in Section 5. In order to exhaustively search the space of plans, and to see the impact of bushy search in isolation, these experiments were run *without* the distance-based heuristic enabled. We allocated a 1h limit to the planner.

Our bushy search looks for left-deep plans as an initialization phase, and thus it generally finds a plan at least as good as the best left-deep plan, if given the time to complete this initialization.

The results show that allowing for bushy plans *can* lead to large savings. For two thirds of our queries, the planner managed to find plans within the allocated time. Moreover, searching for bushy plans pays off with a cost reduction of about one order of magnitude in half of those cases, while other plans were similar to the best left-deep ones.

However, our current bushy search is still quite expensive, often leading to long planning times, reflecting the intrinsic combinatorial difficulty in searching for bushy plans in queries that when, expanded by constraints, may have $>$10 atoms. For the seven remaining answerable queries, the search for bushy plans timed out without yielding any solution. This was partially related to the fact that the experiment did not apply the distance-based heuristic. Under this setting, even a left-deep plan could not be found for Q_{11}, our most expensive query. The remaining queries all use the lineitem relation. In our schema, this relation only has limited access and is 3 steps away from any free access relation in the "dependency graph" that relates tables if they are connected by a dependency. Therefore, a large reasoning effort is required before lineitem is determined to be accessible, an effort that is compounded when bushy search is employed.

An example of the benefits for medium-sized queries is query Q_3, asking for the names and account balance of US suppliers along with the costs of their supplied parts, i.e., $\{suppname, actbal, suppcost \mid partsupp(partkey, suppkey, supp-cost, \ldots) \wedge supplier(suppkey, suppname, nationkey, actbal, \ldots) \wedge nation(nationkey, \text{'US'}, \ldots)\}$. The best left-deep plan found is given by $P = \overrightarrow{\bowtie}(\overrightarrow{\bowtie}(\overrightarrow{\bowtie}(region_nation(nationkey, \text{'US'}, \ldots), part(partkey, \ldots)), supplier(suppkey, suppname, nationkey, actbal, \ldots)), partsupp(\underline{partkey, suppkey}, suppcost, \ldots))$ and has cost 4.96E7. The underlined attributes *partkey* and *suppkey* correspond to inputs for the relation *partsupp*. Note the presence of the view *region_nation* and of the relation *part* in P. Neither of these appear explicitly in Q_3, but the key foreign-key dependencies from *partsupp* to *part* and from *region* to *nation*, as well as the *region_nation* view definition enable us to retrieve the data related to *partsupp* and *nation* from *part* and *region_nation*, respectively. The execution of this plan involves a cartesian product between *region_nation* and *part*, then a traditional join between $\overrightarrow{\bowtie}(region_nation, part)$ and *supplier* and, finally, a dependent join with *partsupp* using the attributes *partkey* and *suppkey*.

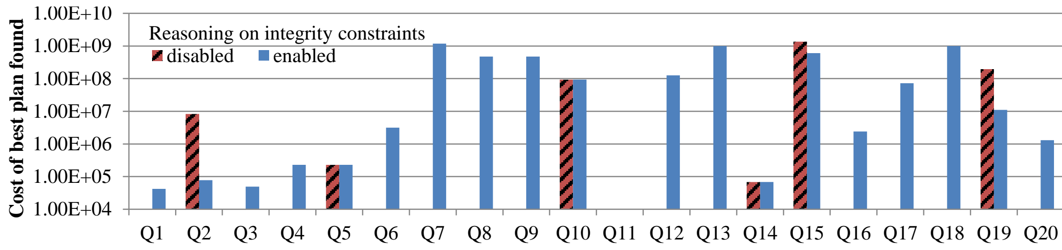

Figure 2: Cost of the best plan with and without reasoning enabled.

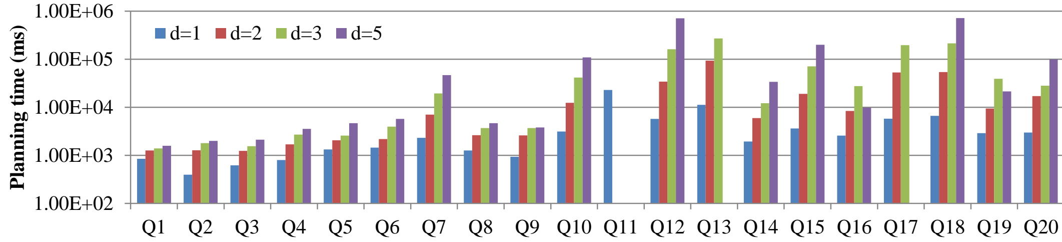

Figure 3: Planning time with varying d

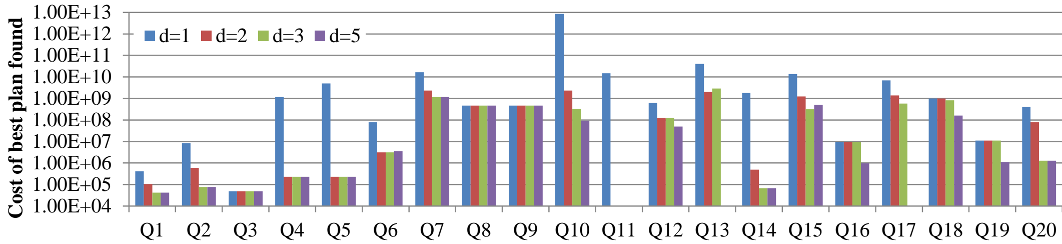

Figure 4: Cost of the best plan with varying d

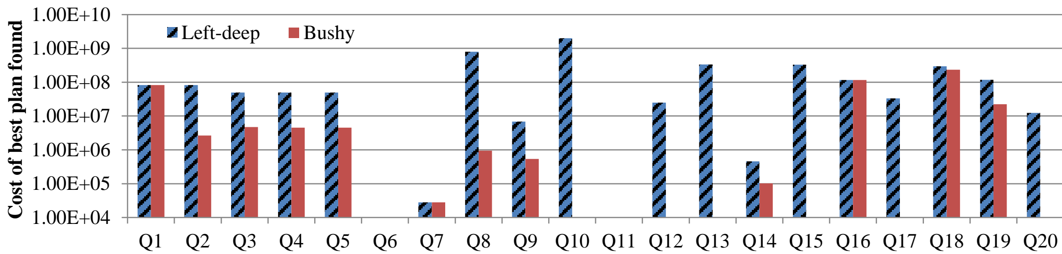

Figure 5: Comparison of cost between left-deep and bushy plans.

The best bushy plan found is given by $P' = \overrightarrow{\bowtie}(\overrightarrow{\bowtie}(region_nation(nationkey, \text{'US'}, \ldots), supplier$ $(suppkey, suppname, nationkey, actbal, \ldots)), \overrightarrow{\bowtie}(part(partkey, \ldots),$ $partsupp(\underline{partkey, suppkey}, suppcost, \ldots)))$ and has cost one order of magnitude lower than the cost of P (4.72E6). This plan performs a traditional join between *region_nation* and *supplier* and then a dependent join between the previously joined data and the output of $\overrightarrow{\bowtie}(part(partkey, \ldots), partsupp(partkey, suppkey, suppcost, \ldots))$.

6.5 Impact of join-order-aware cost-based plan search

Recall that we search for efficient implementations at the plan level, judging cost with a call to a cost function. In particular, we do not identify better plans with some syntactic requirement on a query, such as minimality of the number of query atoms. Our last experiment compares the plans found by our system with those found by a minimal-rewriting approach, such as the chase and backchase of [17, 14, 11]. We mimicked the minimal-rewriting query-based approach (MR below) simply by setting our cost function to be the number of atoms, and then randomly re-ordering the atoms in the resulting plan. Note that, in cases where MR found multiple minimal rewritings, we randomly chose one of them. In evaluating our system and the minimal-rewriting-based one, we used the PostgreSQL plan analyser tool EXPLAIN to evaluate the cost. As our system produces physical plans, in contrast to MR which returns logical rewritings, we translated the plans or rewriting found by our system and MR, respectively, to SQL queries, which were subsequently supplied to PostgreSQL for analysis. We used a schema featuring 12 views with free access and made the base relation inaccessible except region, nation and supplier. All queries are answerable using the accessible tables: we first added materialized views for every pair connected by a key-foreign key constraint, and then added one view for every triple of relations for which there exists a single "hub" relation having key-foreign key constraints with the two other relations, with the remaining relations not being associated via a key foreign-key dependency. In this experiment, our system was restricted to search for left-deep plans.

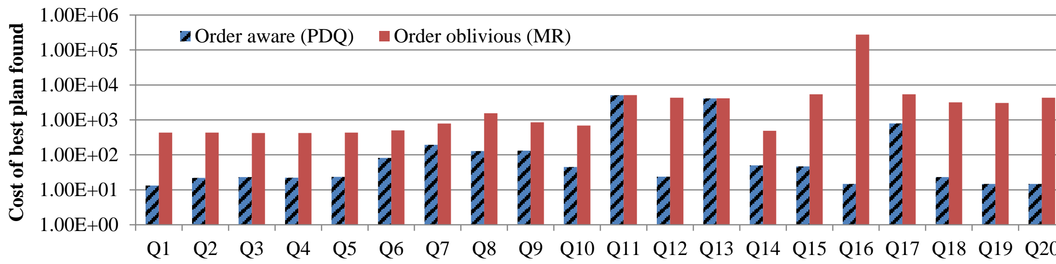

Figure 6: Comparison of costs between order-aware and order-oblivious plans.

The results are presented in Figure 6. In most of the cases, we can observe improvements of several orders of magnitude. For example, Q_2, asks for the manufactures of the parts whose available quantity is 10 units:

$$\{\langle manufacture\rangle \mid partsupp(partkey, partname, 10, \dots) \wedge part(partkey, manufacture, \dots)\}$$

Some of the input-free-access views that our schema has are part_partsupp, supplier_partsupp, and nation_supplier, which are the natural joins of the corresponding tables. It is clear that our query can be directly answered by accessing the part_partsupp view. This is actually the minimal rewriting returned by MR. For this rewriting, PostgreSQL produces a physical plan that, first, performs a sequential scan on part_partsupp and, second, filters out the tuples with available quantity $\neq 10$. The total cost of these operations is estimated by PostgreSQL at 433.00. However, we could answer that query 20 times faster approximately by first accessing the nation_supplier view, then the supplier_partsupp view and finally the part_partsupp view. In particular, the physical plan that PostgreSQL produces joins the tuples of nation_supplier with the ones of supplier_partsupp, after performing, first, a sequential scan on nation_supplier and, second, an indexed scan on supplier_partsupp filtering out the tuples with available quantity $\neq 10$. The total cost of the above operations is estimated at 13.69. As a final step, it joins the previously derived tuples with the ones acquired after performing an indexed scan on part_partsupp and filtering out the tuples with available quantity $\neq 10$. The overall cost is estimated at 22.13.

7. CONCLUSIONS AND FUTURE WORK

We have presented here a basic framework for creating query plans while accounting for both access patterns and integrity constraints. Our goal was to show how these features can be accommodated as a generalization of traditional query optimization.

We believe that the framework gives an approach to a very broad class of queries and constraints. For example, we have considered here only TGDs, but the basic framework can accommodate arbitrary first-order constraints and queries. In particular, we can extend the framework to key dependencies and more generally, Equality-generating Dependencies (EGDs), requiring only the appropriate extension of the chase procedure to account for EGDs.

Our model of access methods here was extremely simplistic, corresponding to a lookup operation. To account for either realistic DBMS access primitives or for the diversity of APIs available in web sources, one would need to deal with a much richer model, allowing e.g. bulk access, block-oriented access, and iteration through a result set. Likewise, for simplicity in this work we assumed the existence of only a "vanilla" join and dependent join operation, and our algorithms returned only one plan, rather than multiple plans representing different "interesting orders" on outputs. But we believe our algorithms can easily be extended to a more sophisticated operator model.

8. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] V. Bárány, M. Benedikt, and P. Bourhis. Access restrictions and integrity constraints revisited. In *ICDT*, pages 213–224, 2013.

[3] V. Bárány, G. Gottlob, and M. Otto. Querying the guarded fragment. In *LICS*, pages 1–10, 2010.

[4] M. Benedikt, J. Leblay, and E. Tsamoura. PDQ: Proof-driven query answering over web-based data. In *VLDB*, 2014.

[5] M. Benedikt, B. ten Cate, and E. Tsamoura. Generating low-cost plans from proofs. In *PODS*, pages 200–211, 2014.

[6] A. Calì, G. Gottlob, T. Lukasiewicz, and A. Pieris. A logical toolbox for ontological reasoning. *SIGMOD Record*, 40(3):5–14, 2011.

[7] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. *TCS*, 371(3):200–226, 2007.

[8] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[9] O. Duschka, M. Genesereth, and A. Levy. Recursive query plans for data integration. *J. Log. Prog.*, 43(1):49 – 73, 2000.

[10] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, pages 311–322, 1999.

[11] I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*, pages 1015–1026, 2014.

[12] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *TODS*, 25(1):43–82, 2000.

[13] C. Li and E. Chang. Answering queries with useful bindings. *TODS*, 26(3):313–343, 2001.

[14] M. Meier. The backchase revisited. *VLDB J.*, 23(3):495–516, 2014.

[15] A. Nash, L. Segoufin, and V. Vianu. Views and queries: Determinacy and rewriting. *TODS*, 35(3):21:1–21:41, 2010.

[16] A. Onet. The chase procedure and its applications in data exchange. In *Digital Enterprise and Information Systems (DEIS)*, pages 1–37, 2013.

[17] L. Popa. *Object/Relational Query Optimization with Chase and Backchase*. PhD thesis, U. Penn., 2000.

[18] D. Toman and G. Weddell. *Fundamentals of Physical Design and Query Compilation*. Morgan Claypool, 2011.