Queries with External Predicates

Paolo Guagliardo
University of Edinburgh, UK

Leonid Libkin
RelationalAI and CNRS, France
University of Edinburgh, UK

Victor Marsault
LIGM, Université Gustave Eiffel, CNRS, France

Wim Martens
University of Bayreuth, Germany
RelationalAI, USA

Filip Murlak
University of Warsaw, Poland

Liat Peterfreund
The Hebrew University of Jerusalem, Israel

Cristina Sirangelo
Université Paris Cité, CNRS, IRIF, F-75013, Paris, France

Abstract

Real-life query languages feature external predicates such as user-defined functions or built-in arithmetic and string operations. These predicates are often infinite, potentially leading to unsafe or non-computable queries. To overcome this, traditional languages such as SQL, put significant syntactic restrictions on the use of external predicates. These restrictions have been relaxed in a number of modern query languages, each doing it in their own way. Our goal therefore is to provide a theoretical basis for querying with external predicates. To this end, we formalize queries with external predicates based on the notion of access patterns. We develop a suitable evaluation model, based on Turing machines with oracles, and tailor the classical notion of query safety to it. Since query safety is undecidable in general, we can only produce sufficient conditions for guaranteeing safety. We do so by developing an inference system to derive safety and computability for relational algebra, first-order logic, as well as for a language that combines them both.

Keywords and phrases External predicates, Query safety, Computational model, Oracles, Infinite predicates, Access patterns, Relational algebra, First-order logic

Digital Object Identifier 10.4230/LIPIcs.ICDT.2025.19

Funding We are grateful to RelationalAI for funding this work.
Leonid Libkin: ANR Project VeriGraph, ANR-21-CE48-0015
Filip Murlak: Poland's National Science Centre grant 2018/30/E/ST6/00042
Liat Peterfreund: Israel Science Foundation 2355/24

1 Introduction

Although first-order logic and relational algebra form a beautiful theoretical basis for query languages, most real-life queries go beyond their scope. For example, none of the TPC-H and TPC-DS benchmark queries, designed to represent real-life workloads, is a pure relational algebra query. One of the key differences is that real-life query languages allow the use of *externally defined predicates*, which can conceptually correspond to *infinite* relations and which queries can only access in a restricted way, for example, using *access patterns*, either explicitly specified or implicit and syntactically enforced.

A very basic example of an external predicate is the addition on integers: `Add` $= \{(a,b,c) \in \mathbb{Z}^3 \mid a+b=c\}$. Indeed, SQL allows writing `SELECT R.a + S.b FROM R,S`, which implicitly gives access to `Add`. This use of addition is restricted, however, by providing an access pattern to `Add` that requires the first two arguments in order to produce the third. Several modern query languages such as Rel [28], Soufflé [19], and .QL [5] are less restrictive and allow much more general access, and even explicit handling of infinite predicates. For example, it is possible in Rel to write the aforementioned SQL query as

$$\texttt{def}\ q(z) \quad : \quad \texttt{exists}(\ (x,y) \mid\ R(x)\ \texttt{and}\ S(y)\ \texttt{and}\ \texttt{Add}(x,y,z)\ ).$$

Here, `Add` is treated just like any database relation, which means that we could also have returned all y such that $\texttt{exists}(\ (x,z) \mid R(x)\ \texttt{and}\ S(z)\ \texttt{and}\ \texttt{Add}(x,y,z)\ )$, which would *not* have been possible in SQL. This more flexible use of external (potentially infinite) predicates in modern languages brings new challenges. For example, we need to be able to detect cases like

$$\texttt{def}\ q'(x,y) \quad : \quad \texttt{exists}(\ (z) \mid\ R(z)\ \texttt{and}\ \texttt{Add}(x,y,z)\ ) \qquad \text{or}$$
$$\texttt{def}\ q''(z) \quad : \quad \texttt{exists}(\ (x,y) \mid\ R(x)\ \texttt{and}\ S(y)\ \texttt{and not}\ \texttt{Add}(x,y,z)\ )$$

which would return infinite results. Notice that q and q' are very close syntactically (they both are conjunctive queries without self-joins); this indicates that syntactic restrictions alone will not suffice.

In practice, the questions we described using `Add` occur for *user-defined functions (UDFs)*, which can provide arbitrary, externally programmed code that can be invoked when querying a database. The exact details vary depending on the concrete DBMSs, but in general such extensions of basic querying facilities provide the interface `CREATE FUNCTION <name>(<parameter-list>) RETURNS <return-type>`. The return type could be a scalar or a table. Queries can then use subexpressions of the form `name(...)`. Upon execution, when provided with concrete parameters, the code for the UDF is inlined in the query execution code. A naive example of a UDF is `Add(x,y)` which simply returns `x + y`; another example is `CountOccurrences(word,text)` that returns the number of occurrences of `word` in `text`. A more complex example is `PathLen(u,v,l)` that returns *true* iff there is a path from node `u` to node `v` of length `l` in a graph stored in the database (see Example 4).

In this paper, we model UDFs as external predicates with *access patterns* [12, 13, 14, 22, 23, 26]. Access patterns provide information about parameters that need to be specified in order to retrieve tuples from a relation corresponding to the predicate. We focus on automatic detection of queries with external predicates that are *computable*, aiming at syntactic criteria for termination. Based on these, a DBMS can immediately discard non-computable queries rather than letting the user wait for a timeout.

Our contribution is twofold. First, we formalize abstract (language-agnostic) queries with external predicates and define what it means for such a query to be computable. We propose a model of computation that is *uniform* with respect to external predicates: the evaluation algorithm accesses them via oracles. This reflects how UDFs are handled in execution engines—as black boxes. Indeed, while the result of a query depends on the semantics of the UDFs it uses, the evaluation algorithm does not: all the execution engine does is call some procedures at some points during query evaluation and continue with their results. Next, in preparation for the second part, we characterize computability in terms of an *effective* variant of the well-known notion of *safety* [8, 11, 21, 27, 29, 30] and a complementary notion we call *testability*. Effective safety ensures not only that the output of the query is finite, but also that a finite superset of the output can be effectively produced; testability amounts to the decidability of the membership problem associated with the query, meaning that we can decide if a given tuple is in the output.

Second, we look at concrete query languages. Relational query languages are often

presented in a dichotomic way: there are declarative languages (such as relational calculus) that are translated into procedural languages (such as relational algebra) that in turn are evaluated by a query engine. The reality is more complicated: languages like SQL have a declarative component (basic `SELECT-FROM-WHERE` statements) and a procedural one (set operations `UNION`, `INTERSECT`, `EXCEPT`, explicit `JOIN` . . . `ON` in `FROM`). Another such language is Rel [28], where the user can turn formulas into relations using comprehension and, conversely, check membership of tuples with variables in complex relational expressions. To reflect this, we use a formalism called FO/RA, obtained by combining first order logic (FO) and relational algebra (RA) by means of *comprehension* and *membership* tests to go between the two.
In the presence of external predicates, even simple relational algebra expressions are not necessarily computable. In fact, safety, effective safety, and testability are all undecidable for FO, RA, and FO/RA. Still, it is desirable to be able to identify a large class of computable queries. To this end we develop an effective rule system for deriving computability of queries with external predicates, which works by independently propagating effective safety and testability through the syntactic representation of queries. Compared to a heuristic algorithm, an effective inference system offers enhanced explainability, transparency, and extensibility:

- computability of a query can be witnessed by a derivation;
- the execution engine can guarantee to handle all derivably computable queries and the class of handled queries can be precisely described to users in terms of inference rules;
- new language constructs can be handled by including additional rules, as illustrated in our inference system for FO/RA, which essentially combines inference systems for FO and RA, with additional rules handling comprehension and membership tests.

Our system is incomplete (which is the case for every effective inference system for deriving an undecidable property), but we provide some basic *relative* completeness results, showing that our system can capture computability (up to query equivalence) in some restricted cases.

While our actual target is FO/RA, we begin by looking at RA and FO separately, which allows introducing necessary ingredients gradually. Why consider both FO and RA, if they are well known to be equivalent? The classical equivalence of RA and FO [2, Chapter 6] relies on the active domain semantics of FO. In the presence of external predicates, however, we can no longer assume it, since external predicates can have infinite domains. Moreover, translation-based approaches to safety are generally problematic. One reason is that safety can get "lost in translation" due to the inherent incompleteness of safety inference systems, which manifests itself as strong syntax-dependence. Another reason why relying on translations is a bad idea, is that, in a practical scenario, (un)safety errors would lead to cryptic error messages, referring to translated queries, not the originals. Safety rules should therefore cover all constructs of the real language.

Our work has both immediate and long term practical relevance. Newly developed languages whose syntax allows unsafe queries currently attempt to rule them out using bespoke solutions that change with every new compiler release. We provide them with a systematic approach that can be used for base-line guarantees of safety and computability. In longer term, our results could be used to relax the strict syntactic rules regulating the use of UDFs in SQL. We envision users specifying access patterns for UDFs and the system detecting unsafe usage of UDFs in queries.

Organization. Computability theory for queries with external predicates is developed in Section 2. In Sections 3 and 4 we develop inference systems for RA and FO, and in Section 5 we combine them into an inference system for FO/RA. We describe related work in Section 6 and conclude in Section 7.

2 Framework

We assume a countably infinite set $\mathbb{V}$ of *values* (containing integers, floats, etc.) from which we will take the data values in databases. Because we deal with relational algebra, first-order logic, and a formalism that combines the two, it is convenient to work with a generalized notion of tuple, that captures both ordinary tuples and valuations of variables. A *tuple schema* is a finite set; elements of a tuple schema are called *fields*. For a tuple schema $\mathcal{F}$, an *$\mathcal{F}$-tuple* is simply a function $\mathcal{F} \to \mathbb{V}$. We write $\mathbb{V}^{\mathcal{F}}$ for the set of all $\mathcal{F}$-tuples. An $\mathcal{F}$-relation is a (possibly infinite) set $R \subseteq \mathbb{V}^{\mathcal{F}}$ of $\mathcal{F}$-tuples. There is exactly one $\varnothing$-tuple and thus there are exactly two $\varnothing$-relations: one that contains the $\varnothing$-tuple, and one that does not. We say that an $\mathcal{F}_1$-tuple α_1 *extends* an $\mathcal{F}_2$-tuple α_2 if $\mathcal{F}_1 \supseteq \mathcal{F}_2$ and $\alpha_1(f) = \alpha_2(f)$ for all $f \in \mathcal{F}_2$. We then also say that α_2 is *a restriction* of α_1 (to $\mathcal{F}_2$).

For $k \in \mathbb{N}$, we denote the set $\{1, \dots, k\}$ as $[k]$. We refer to $[k]$-tuples as *k-ary tuples*, and identify $\alpha \in \mathbb{V}^{[k]}$ with $\big(\alpha(1), \dots, \alpha(k)\big) \in \mathbb{V}^k$ where $\mathbb{V}^k$ is the k-ary Cartesian product of $\mathbb{V}$. We also use $\mathcal{F} \subseteq [k]$ to represent "partial" k-ary tuples. We use $\mathbb{V}^{[k]}$ and $\mathbb{V}^k$ interchangeably. We let $\mathbb{V}^* = \bigcup_{i=0}^{\infty} \mathbb{V}^k$. A *$k$-ary relation* or a *relation of arity k* is a (possibly infinite) set of k-ary tuples.

2.1 Queries with External Predicates

We work with two kinds of predicates (or *relation names*), *database predicates* coming from an infinite set $\mathbb{D}$, and *external predicates* coming from an infinite set $\mathbb{E}$, disjoint from $\mathbb{D}$. We assume a fixed function $\mathsf{ar} : \mathbb{D} \cup \mathbb{E} \to \mathbb{N}$ that assigns to each predicate R its *arity* $\mathsf{ar}(R)$.

Database predicates are the names of the (finite) relations we store in databases. A *database schema* is a finite subset $\mathcal{S} \subseteq \mathbb{D}$. A *database D over schema $\mathcal{S}$* is a function that maps each database predicate $R \in \mathcal{S}$ to a finite relation $D(R) \subseteq \mathbb{V}^{\mathsf{ar}(R)}$. We write $\mathrm{DB}(\mathcal{S})$ for the set of all databases over $\mathcal{S}$.

External predicates correspond to relations that are computed on demand rather than stored in the database: built-in predicates such as arithmetical functions, or user-defined predicates such as stored procedures in SQL databases. Crucially, they may be infinite and we have to specify how they are accessed by the database system. We capture this through the notions of *access patterns* and *access schemas.*

▸ **Definition 1** (Access Pattern). *For a finite set $\mathcal{F}$, an* access pattern over $\mathcal{F}$ *is a pair of sets $I, O \subseteq \mathcal{F}$, written $I \rightsquigarrow O$. A k-*ary access pattern, *for $k \in \mathbb{N}$, is an access pattern over $[k]$.*

For a (possibly infinite) $\mathcal{F}$-relation $R \subseteq \mathbb{V}^{\mathcal{F}}$, we write $\mathsf{Access}_R^{I \rightsquigarrow O}$ for the function that maps an I-tuple α to the set of O-tuples β such that there is an $\mathcal{F}$-tuple $\gamma \in R$ that extends both α and β. Relation R supports *$I \rightsquigarrow O$ if $\mathsf{Access}_R^{I \rightsquigarrow O}(\alpha)$ is finite for every I-tuple α and moreover the function $\mathsf{Access}_R^{I \rightsquigarrow O}$ is computable.*

When O is a singleton $\{o\}$ we usually write $I \rightsquigarrow o$ instead of $I \rightsquigarrow \{o\}$, and similarly for I. Observe that $R \subseteq \mathbb{V}^{\mathcal{F}}$ supports the *trivial* access pattern $\mathcal{F} \rightsquigarrow \varnothing$ iff the membership problem for R is decidable. On the other hand, $R \subseteq \mathbb{V}^{\mathcal{F}}$ supports $\varnothing \rightsquigarrow \mathcal{F}$ iff R is finite. Finally, every finite $\mathcal{F}$-relation supports all access patterns over $\mathcal{F}$.

▸ **Definition 2** (Access Schema). *An* access schema *is a finite-domain partial function $\mathcal{E}$ over $\mathbb{E}$ that maps an external predicate L of arity k to a set $\mathcal{E}(L)$ of k-ary access patterns. By a slight abuse of notation, we write $L \in \mathcal{E}$ if $\mathcal{E}$ is defined on L.*

▸ Remark 3. In this paper we will only look at cases for which the membership problem for each external predicate is decidable, hence we implicitly assume that access schemas always contain the access pattern $[\mathsf{ar}(L)] \rightsquigarrow \varnothing$ for every external predicate L.

▸ **Example 4.** Consider a database with a unary relation N and a binary relation E storing nodes and edges of a graph. Let S be of the set triples (u, v, ℓ) such that there is a path from u to v of length ℓ in the stored graph. Although S does not support $\{1,2\}{\leadsto}3$ (there might be infinitely many paths of different lengths from some u to some v), it supports $\{1,2,3\}{\leadsto}\varnothing$ since it is easy to test if there is a path of length ℓ from node u to node v. In fact, S supports all access patterns $I{\leadsto}O$ with $3 \in I$. That said, a system may choose to not provide the access pattern $3{\leadsto}\{1,2\}$ because it might be too expensive due to its inherent quadratic complexity. The access pattern $1{\leadsto}2$ corresponds to allowing to return all nodes reachable from a given node. This access pattern is also supported by S (despite using a strict subset of the three positions of S) and is less costly computationally.

User defined functions often depend on the content of the database: in Example 4, the set S depends on the stored graph. However, to keep our framework simple, we shall disregard such dependencies, just as we disregard integrity constraints such as $E \subseteq N \times N$. This does not limit the scope of our results because our algorithms and inference systems are designed to work uniformly with all databases and all interpretations of external predicates, including the intended combinations. While incorporating such dependencies into the model might in principle allow obtaining stronger results, we believe this is not a priority.

▸ **Definition 5** ($\mathcal{E}$-Interpretation)**.** *An* interpretation λ *is a partial function that maps external predicates to possibly infinite relations of matching arity, i.e* $\lambda(L) \subseteq \mathbb{V}^{\mathsf{ar}(L)}$*, for* $L \in \mathcal{E}$*. It is an* $\mathcal{E}$-interpretation *if, for each* $L \in \mathcal{E}$*, the relation* $\lambda(L)$ *is defined and supports all access patterns in* $\mathcal{E}(L)$*. We write* $\Lambda(\mathcal{E})$ *for the set of all* $\mathcal{E}$*-interpretations.*

Note that $\mathsf{Access}_{\lambda(L)}^{I\leadsto O}$ is finite for every $\mathcal{E}$*-interpretation* λ and pattern $I{\leadsto}O$ in $\mathcal{E}(L)$.

Consider Example 4 and let $\mathcal{E}$ be the access schema mapping a name $\mathsf{PathLen} \in \mathbb{E}$ to $\{\{1,2,3\}{\leadsto}\varnothing, \{1,3\}{\leadsto}2, \{2,3\}{\leadsto}1\}$. Then the function that maps $\mathsf{PathLen}$ to the set S in Example 4 is an $\mathcal{E}$-interpretation. As the name suggests, access patterns in $\mathcal{E}(\mathsf{PathLen})$ model how the engine allows one to access predicate $\mathsf{PathLen}$, not only the finiteness and computability of this information. Hence, an $\mathcal{E}$-interpretation λ might be such that $\lambda(\mathsf{PathLen})$ supports some access pattern $I{\leadsto}O \notin \mathcal{E}(\mathsf{PathLen})$, such as $3{\leadsto}\{1,2\}$ in our case.

▸ **Definition 6** (Queries)**.** *A* query over access schema $\mathcal{E}$, database schema $\mathcal{S}$, and output schema $\mathcal{F}$ *is a function* $q : \Lambda(\mathcal{E}) \times DB(\mathcal{S}) \to 2^{\mathbb{V}^{\mathcal{F}}}$*; that is, given an* $\mathcal{E}$*-interpretation and a database over* $\mathcal{S}$*, it returns a (possibly infinite) set of* $\mathcal{F}$*-tuples. We then also call* q *an* $\mathcal{E}$-query*. If* $\mathcal{F} = [k]$*, we say that* q *is* k-ary*. In particular, the output schema of a unary query is* $\{1\}$*.*

▸ **Example 7.** Continuing Example 4, consider an external predicate Rank that computes the rank of a node based on the graph that is encoded in the database relation E. So, Rank is a set of pairs (u, r) where u is a node and r is a value normalized to the interval [0,1] (e.g., PageRank, betweenness, or another centrality score). The query $\mathrm{HR}(x) = \exists r\ \mathsf{Rank}(x, r) \wedge r > 0.8$ determines if a node x has a high rank. The query $N(u) \wedge \mathrm{HR}(u) \wedge (\forall v E(v, u) \Rightarrow \neg\mathrm{HR}(v))$ returns all nodes u that are highly ranked such that all nodes v with edges to u are not (which may be interesting to detect cases of bots trying to artificially boost importance). Notice that, formally, different interpretations λ can be used to associate different meanings to Rank (PageRank, etc.).

2.2 Computability of $\mathcal{E}$-queries

Since external predicates are potentially infinite, we need to define what it means for queries to be *computable*, because infinite objects are not usually part of the input of computational problems. We base our computational model on Turing machines with oracles.

▸ **Definition 8** (Turing Machine with Oracles, adapted from [4]). *A Turing machine with k oracles $\omega_1, \ldots, \omega_k$ is a 3-tape Turing machine with an input tape and two special working tapes, called the* oracle input tape *and the* oracle output tape*, as well as special states* call_i *and* return_i *for every $i \in [k]$. Moreover, the transition table does not have any transitions originating in* call_i*, nor leading to* return_i*.*

Given an interpretation of each oracle name ω_i as a function Ω_i from words over the tape alphabet to words over the tape alphabet, the machine behaves as an ordinary Turing machine except when it enters a state call_i *for some $i \in [k]$. Whenever this happens, the current content of the oracle output tape is replaced with $\Omega_i(u)$ where u is the current content of the oracle input tape, and the machine moves to state* return_i*. For a deterministic machine M with k oracles, we write $M^{\Omega_1,\ldots,\Omega_k}(w)$ for the output computed by M with oracle functions $\Omega_1, \ldots, \Omega_k$ over input w.*

From now on, we assume (as is standard in descriptive complexity and finite model theory) that we have some fixed encoding of tuples and relations as words over the tape alphabet (see examples in [2, 16, 24]). With this, we can speak of Turing Machines taking a database or a tuple as input and returning a set of tuples.

▸ **Definition 9** ($\mathcal{E}$-algorithm). *For an access schema $\mathcal{E}$, by an $\mathcal{E}$-algorithm $\mathcal{A}$ we mean a deterministic Turing machine M with an oracle named $L_{I \rightsquigarrow O}$ for each $L \in \mathcal{E}$ and $I \rightsquigarrow O$ in $\mathcal{E}(L)$. For an $\mathcal{E}$-interpretation λ, by $\mathcal{A}^\lambda$ we denote the function $M^{\bar{\Omega}}$ where $\bar{\Omega}$ interprets each oracle name $L_{I \rightsquigarrow O}$ as $\mathsf{Access}^{I \rightsquigarrow O}_{\lambda(L)}$. We say that $\mathcal{A}$* computes *a function $f : \Lambda(\mathcal{E}) \times X \to Y$, for some sets X, Y, if $\mathcal{A}^\lambda(x) = f(\lambda, x)$ for every $\mathcal{E}$-interpretation λ and $x \in X$. We say that f is* uniformly computable *if there is an $\mathcal{E}$-algorithm that computes it. We then also say that $f(\lambda, x)$ is* uniformly computable from input *x.*

The above definition applies to $\mathcal{E}$-queries which are functions $q : \Lambda(\mathcal{E}) \times \mathrm{DB}(\mathcal{S}) \to 2^{\mathbb{V}^{\mathcal{F}}}$; thus it specifies when $\mathcal{E}$-queries are uniformly computable. Later we also apply it to functions $f : \Lambda(\mathcal{E}) \times \mathrm{DB}(\mathcal{S}) \times \mathbb{V}^I \to 2^{\mathbb{V}^{\mathcal{F}}}$ for $I \subseteq \mathcal{F}$ that additionally take an I-tuple as input, which corresponds to letting $X = \mathrm{DB}(\mathcal{S}) \times \mathbb{V}^I$.

Note that the notion of computability provided by $\mathcal{E}$-algorithms is *uniform* with respect to $\mathcal{E}$-interpretations. That is, not only is the function computable for each interpretation, but it is computed by the same algorithm given by a single Turing machine with oracles. As such, it corresponds to the intuition of a program that can call external functions: the program itself is fixed and independent from the external functions, whereas the output of the program depends on the semantics of the external functions. The interfaces to the external functions are given by $\mathcal{E}$. Continuing Example 7, an $\mathcal{E}$-algorithm would iterate through all nodes in the relation E of edges, use oracle calls to determine their rank, and then return all nodes u with high rank for which all its in-neighbors do not have high rank.

To round off the discussion of computability of $\mathcal{E}$-queries, let us sketch a simple connection to query evaluation in the classical setting. If λ is a *finite* $\mathcal{E}$-interpretation, i.e., $\lambda(L)$ is finite for all $L \in \mathcal{E}$, it can be materialized and stored in the database. In general this is not possible, but the following proposition (a variant of a folklore property of Turing machines with oracles) shows that, for uniformly computable queries, it suffices to store a sufficiently large finite chunk of each external predicate.

▸ **Proposition 10.** *Let q be a uniformly computable $\mathcal{E}$-query over database schema $\mathcal{S}$ and output schema $\mathcal{F}$. For every database D over $\mathcal{S}$ and every $\mathcal{E}$-interpretation λ there is a finite $\mathcal{E}$-interpretation λ' such that $\lambda'(L) \subseteq \lambda(L)$ for all $L \in \mathcal{E}$, and $q(\lambda, D) = q(\lambda', D)$. Moreover, some such λ' is uniformly computable from D.*

2.3 Safety of $\mathcal{E}$-queries

A prerequisite for the computability of $\mathcal{E}$-queries is *safety*, i.e., finiteness of the output.

▸ **Definition 11** (Safety). *An $\mathcal{E}$-query q is* safe *if $q(\lambda, D)$ is finite for every $\mathcal{E}$-interpretation λ and database D.*

However, a safe $\mathcal{E}$-query needs not be uniformly computable (see Example 33 in the appendix). Our goal is to derive uniform computability for queries expressed in formalisms such as relational algebra or first order logic. For this we need a notion that is flexible enough to propagate up the syntax tree of the query and captures not only finiteness but also computability. With these postulates in mind we introduce *relative effective safety*. To keep the definition concise, we use two auxiliary notions. For an $\mathcal{E}$-query q over database schema $\mathcal{S}$ and output schema $\mathcal{F}$, and access pattern $I \rightsquigarrow O$ over $\mathcal{F}$, we define $q^{I \rightsquigarrow O} : \Lambda(\mathcal{E}) \times \mathrm{DB}(\mathcal{S}) \times \mathbb{V}^I \to 2^{\mathbb{V}^O}$ such that $q^{I \rightsquigarrow O}(\lambda, D, \alpha) = \mathsf{Access}^{I \rightsquigarrow O}_{q(\lambda,D)}(\alpha)$ for all $\lambda \in \Lambda(\mathcal{E})$, $D \in \mathrm{DB}(\mathcal{S})$, and $\alpha \in \mathbb{V}^I$. By an *overapproximation* of a function $f : X \to 2^Y$ we mean any function $f' : X \to 2^Y$ such that $f(x) \subseteq f'(x)$ for all $x \in X$.

▸ **Definition 12** (Effective safety). *Let q be a $\mathcal{E}$-query over output schema $\mathcal{F}$ and let $I \subseteq \mathcal{F}$. The query q is* effectively safe relative to I *if there is a uniformly computable overapproximation of $q^{I \rightsquigarrow \mathcal{F}}$. When $I = \varnothing$, we simply say that q is* effectively safe.

Intuitively, relative effective safety means that, given values for all fields from I, we can uniformly compute, from an input database, a finite upper bound on the set of query answers having the specified values in all fields from I. In particular, if a query is effectively safe, then not only do we know that it is safe but we can also compute an upper bound on the set of answers on a given database. For instance, the query in Example 7 is trivially effectively safe because the set of nodes in N is always an overapproximation of the output of the query.

In Definition 12 we let the algorithm compute an overapproximation of $q^{I \rightsquigarrow \mathcal{F}}$, rather than $q^{I \rightsquigarrow \mathcal{F}}$ itself, in order to simplify propagation up the syntax tree, as illustrated below.

▸ **Example 13.** Consider effectively safe unary $\mathcal{E}$-queries q_1, q_2, an arbitrary unary $\mathcal{E}$-query q_3, and assume that $\{1,2\} \rightsquigarrow 4$ belongs to $\mathcal{E}(L)$, for some $L \in \mathbb{E}$ with $\mathsf{ar}(L) = 4$. Then, the unary $\mathcal{E}$-query q, below, is also effectively safe.

$$q(\lambda, D) = \left\{x_4 \;\middle|\; \exists x_1 \in q_1(\lambda, D) \;\; \exists x_2 \in q_2(\lambda, D) \;\; \exists x_3 \in q_3(\lambda, D) \quad (x_1, x_2, x_3, x_4) \in \lambda(L)\right\}$$

Indeed, by the hypotheses, there are two $\mathcal{E}$-algorithms $\mathcal{A}_1, \mathcal{A}_2$ that compute overapproximations of $q_1^{\varnothing \rightsquigarrow 1}$ and $q_2^{\varnothing \rightsquigarrow 1}$. From them, one may build the following $\mathcal{E}$-algorithm $\mathcal{A}$. For every a_1 returned by $\mathcal{A}_1$ and a_2 returned by $\mathcal{A}_2$, call the $L_{\{1,2\} \rightsquigarrow 4}$ oracle on input (a_1, a_2). Return the union of all sets returned by these (finitely many) calls. It can be verified that $\mathcal{A}$ computes an overapproximation of $q^{\varnothing \rightsquigarrow 1}$.

Note that our reasoning needs no assumption on q_3, and is valid only because we work with overapproximations. Indeed, one cannot uniformly compute $q^{\varnothing \rightsquigarrow 1}$ from $q_1^{\varnothing \rightsquigarrow 1}$ and $q_2^{\varnothing \rightsquigarrow 1}$.

Ultimately, however, we need the exact set of answers. To facilitate this, we introduce a complementary notion of testability, which will be derived separately.

▸ **Definition 14** (Testability). *An $\mathcal{E}$-query q over output schema $\mathcal{F}$ is* testable *if $q^{\mathcal{F} \rightsquigarrow \varnothing}$ is uniformly computable.*

Thus, q is testable if checking whether $\alpha \in q(\lambda, D)$ is uniformly computable from D and α. While effective safety alone does not guarantee uniform computability (see Example 33 in the appendix), together with testability it captures uniform computability exactly.

▸ **Proposition 15.** *An $\mathcal{E}$-query q is uniformly computable iff it is effectively safe and testable.*

Using overapproximations allows us to reason about the effective safety of single fields rather than the whole query, which further improves propagation up the syntax tree.

▸ **Definition 16** (Relative effective safety of fields)**.** *Let q be an $\mathcal{E}$-query over output schema $\mathcal{F}$ and let $I \subseteq \mathcal{F}$. A field $o \in \mathcal{F}$ is* effectively safe in q relative to I *if there is a uniformly computable overapproximation of $q^{I \rightsquigarrow o}$.*

For instance, if $L \in \mathcal{E}$ and q is the $\mathcal{E}$-query mapping each (λ, D) to the relation $\lambda(L)$ then, for each $I \rightsquigarrow O$ in $\mathcal{E}(L)$, each field $o \in O$ is effectively safe in q relative to I.

▸ **Proposition 17.** *Let q be an $\mathcal{E}$-query over output schema $\mathcal{F}$ and let $I \subseteq \mathcal{F}$. Query q is effectively safe relative to I iff each field in $\mathcal{F}$ is effectively safe in q relative to I.*

In the remainder of the paper we use the criterion in Proposition 17 as an equivalent definition of (relative) effective safety.

3 Relational Algebra with External Predicates

In this section, we consider Relational Algebra (RA) in the unnamed perspective, adjusted to handle external predicates. For an access schema $\mathcal{E}$ and a database schema S, the syntax of $\mathcal{E}$-expressions (of RA) over $\mathcal{S}$ is given by the grammar

$$E ::= R \mid L \mid \alpha \mid E \cup E \mid E - E \mid E \times E \mid \sigma_{i \doteq j}(E) \mid \pi_s(E)$$

where $R \in \mathcal{S}$, $L \in \mathcal{E}$, $\alpha \in \mathbb{V}^*$, $i, j \in \mathbb{N}$, and $s \in \mathbb{N}^*$ (note that s can have repetitions). In what follows, by an RA expression we mean an $\mathcal{E}$-expression of RA over $\mathcal{S}$ for some access schema $\mathcal{E}$ and some database schema $\mathcal{S}$. The *semantics* of an $\mathcal{E}$-expression E over $\mathcal{S}$, written $[\![E]\!]$, is a function that maps an $\mathcal{E}$-interpretation λ and a database D over $\mathcal{S}$ to a set of tuples, defined as follows. For $R \in \mathcal{S}$ and $L \in \mathcal{E}$ we have $[\![R]\!](\lambda, D) = D(R)$ and $[\![L]\!](\lambda, D) = \lambda(L)$. The rest of the inductive definition is as usual [2]. We denote by $\mathsf{ar}(E)$ the arity of E (defined as usual) and by $\mathsf{pos}(E)$ the set of positions in E, that is, $\mathsf{pos}(E) = \{1, \dots, \mathsf{ar}(E)\}$. We only allow expressions $E_1 \theta E_2$ with $\theta \in \{\cup, \cap, -\}$ whenever $\mathsf{ar}(E_1) = \mathsf{ar}(E_2)$.

Note that the semantics $[\![E]\!]$ of an $\mathcal{E}$-expression E over $\mathcal{S}$ is a query over access schema $\mathcal{E}$, database schema $\mathcal{S}$ and output schema $\mathsf{pos}(E)$. We call E *testable*, *effectively safe (relative to a set $P \subseteq \mathsf{pos}(E)$ of positions)*, or *uniformly computable*, if so is the query $[\![E]\!]$. Similarly we say that *position i is effectively safe in E relative to P*, if so is field i in the query $[\![E]\!]$; that is, $[\![E]\!]^{P \rightsquigarrow i}$ has a uniformly computable overapproximation. In the following proposition we highlight some basic facts about testability and effective safety of $\mathcal{E}$-expressions.

▸ **Proposition 18.**
1. *Some RA expressions are not testable, and some are not effectively safe.*
2. *All RA expressions without external predicates are testable and effectively safe.*
3. *Testability, safety, and effective safety of RA expressions are undecidable.*

Proof sketch. Concerning item 1, consider an empty database schema $\mathcal{S}$ and an access schema $\mathcal{E}$ with a single binary predicate L such that $\mathcal{E}(L) = \{\{1, 2\} \rightsquigarrow \varnothing\}$. Then, the $\mathcal{E}$ expression $E_0 = L$ is not safe and hence not effectively safe. Concerning testability, we show that there cannot be an $\mathcal{E}$-algorithm $\mathcal{A}$ that uniformly decides whether $a \in [\![E_1]\!](\lambda, D)$ from input (D, a) where $E_1 = \pi_1(L)$. The contradiction arises with two interpretations: $\lambda_1(L) = \varnothing$ and $\lambda_2(L) = \{(a, v)\}$, where v is a value not appearing in any of the finitely many calls of the $L_{\{1,2\}\rightsquigarrow\varnothing}$ oracle during the execution of $\mathcal{A}^{\lambda_1}(D,a)$. The execution of $\mathcal{A}^{\lambda_2}(D,a)$ is identical to the one of $\mathcal{A}^{\lambda_1}(D,a)$, though the first should accept and the second reject.

Item 2 is easy. For item 3, given an RA expression F without external predicates over a schema $\mathcal{S}$, testing if the output of F is empty on each database is undecidable [2, Ch. 8]. Undecidability of (effective) safety is obtained by using expression $E_0 \times F$, and undecidability of testability by expression $E_1 \times F$, where E_0 and E_1 are the expressions from item 1. ◂

3.1 Inference System $\mathsf{IS}_{\mathsf{RA}}$

We now present an inference system $\mathsf{IS}_{\mathsf{RA}}$ for deriving effective safety and testability of RA expressions. Then, we will prove that $\mathsf{IS}_{\mathsf{RA}}$ is sound (Theorem 20), that it can derive effective safety of all expressions without external predicates (Proposition 22), and that derivability of testability and effective safety in the system is decidable (Proposition 21). In $\mathsf{IS}_{\mathsf{RA}}$, we write "$i$ is effectively safe in E relative to P" as $E \vdash P \multimap i$ and "E is testable" as $E\Downarrow$. Moreover, whenever we write $E \vdash P \multimap i$, we implicitly assume that $P \subseteq \mathsf{pos}(E)$ and $i \in \mathsf{pos}(E)$. In rule $\langle 9\rangle$ below, $P_2 + \mathsf{ar}(E_1)$ stands for $\{j + \mathsf{ar}(E_1) \mid j \in P_2\}$. In rules $\langle 14\rangle$ and $\langle 15\rangle$, the tuple s of positions is used as a function $\{1,\dots,|s|\} \to \mathsf{pos}(E)$ and in rule $\langle 21\rangle$, $\mathsf{set}(s)$ is the set of elements in s.

Axiom, weakening, and cut

$$\langle 1\rangle\frac{}{E \vdash \{i\} \multimap i} \qquad \langle 2\rangle\frac{E \vdash P \multimap i}{E \vdash P \cup \{i'\} \multimap i} \qquad \langle 3\rangle\frac{E \vdash P \multimap i \quad E \vdash P' \multimap i'}{E \vdash P \cup (P' - \{i\}) \multimap i'}$$

Atomic RA expressions

$$\langle 4\rangle\frac{}{\alpha \vdash \varnothing \multimap i}\,\alpha \in \mathbb{V}^* \qquad \langle 5\rangle\frac{}{R \vdash \varnothing \multimap i}\,R \in \mathcal{S} \qquad \langle 6\rangle\frac{}{L \vdash I \multimap i}\,L \in \mathcal{E},\ I\rightsquigarrow O \in \mathcal{E}(L),\ i \in O$$

RA operations

$$\langle 7\rangle\frac{E_1 \vdash P_1 \multimap i \quad E_2 \vdash P_2 \multimap i}{E_1 \cup E_2 \vdash P_1 \cup P_2 \multimap i} \qquad \langle 8\rangle\frac{E_1 \vdash P_1 \multimap i}{E_1 \times E_2 \vdash P_1 \multimap i} \qquad \langle 9\rangle\frac{E_2 \vdash P_2 \multimap i}{E_1 \times E_2 \vdash P_2 + \mathsf{ar}(E_1) \multimap i + \mathsf{ar}(E_1)}$$

$$\langle 10\rangle\frac{E_1 \vdash P_1 \multimap i}{E_1 - E_2 \vdash P_1 \multimap i} \qquad \langle 11\rangle\frac{}{\sigma_{i \doteq j}(E) \vdash \{i\} \multimap j} \qquad \langle 12\rangle\frac{}{\sigma_{i \doteq j}(E) \vdash \{j\} \multimap i}$$

$$\langle 13\rangle\frac{E \vdash P \multimap i'}{\sigma_{i \doteq j}(E) \vdash P \multimap i'} \qquad \langle 14\rangle\frac{}{\pi_s(E) \vdash \{i\} \multimap i'}\,s(i) = s(i') \qquad \langle 15\rangle\frac{E \vdash P \multimap i}{\pi_s(E) \vdash P' \multimap i'}\,s(P') = P, s(i') = i$$

Testability

$$\langle 16\rangle\frac{}{\{\alpha\}\Downarrow}\,\alpha \in V^* \qquad \langle 17\rangle\frac{}{R\Downarrow}\,R \in \mathcal{S} \qquad \langle 18\rangle\frac{}{L\Downarrow}\,L \in \mathcal{E} \qquad \langle 19\rangle\frac{E\Downarrow}{\sigma_{i \doteq j}(E)\Downarrow}$$

$$\langle 20\rangle\frac{E_1\Downarrow \quad E_2\Downarrow}{(E_1\ \theta\ E_2)\Downarrow}\,\theta \in \{\cup, -, \times\} \qquad \langle 21\rangle\frac{E\Downarrow \quad E \vdash \mathsf{set}(s) \multimap i \text{ for each } i \in \mathsf{pos}(E) - \mathsf{set}(s)}{\pi_s(E)\Downarrow}$$

Note that atomic RA expressions are effectively safe and testable (Proposition 18.2), hence the rules $\langle 4\rangle$, $\langle 5\rangle$, $\langle 16\rangle$ and $\langle 17\rangle$. Similarly, effective safety for external predicates (rule $\langle 6\rangle$) are implied by the presence of suitable access patterns (Remark 3). Testability for external predicate (rule $\langle 18\rangle$) is unconditional, because the trivial access pattern is always present.

▸ **Example 19.** The inference system we presented is suited to the rather minimalistic syntax we use. To illustrate how the system works, let us consider a derived operation: intersection. It follows directly from the definitions that if position i is safe in expression E relative to positions P, the same is true in expressions $E \cap F$ and $F \cap E$. Hence, if we allowed the $\cap$-operator in the syntax, it would be natural to add the following extra rules.

$$\langle 22\rangle\frac{E_1 \vdash P_1 \multimap i}{E_1 \cap E_2 \vdash P_1 \multimap i} \qquad \langle 23\rangle\frac{E_2 \vdash P_2 \multimap i}{E_1 \cap E_2 \vdash P_2 \multimap i}$$

Can we derive them in our system? Actually, it depends on how we express the intersection operator using the already supported operators. If we take $E_1 \cap E_2 = E_1 - (E_1 - E_2)$, then we can derive rule ⟨22⟩ using an application of rule ⟨10⟩. However, there is no way to derive ⟨23⟩: the only rule applicable to the expression $E_1 - (E_1 - E_2)$ is ⟨10⟩, and it drops all information about the second argument. On the other hand, if we take $E_1 \cap E_2 = \pi_{1,\dots,n}(\sigma_{1=n+1} \dots \sigma_{n=2n}(E_1 \times E_2))$, where n is the arity of both E_1 and E_2, then both rules are derivable. For instance, we give below the derivation tree corresponding to rule ⟨23⟩ in the case where E_1 and E_2 are binary, $P_2 = \{1\}$ and $i = 2$.

$$\langle 15\rangle\dfrac{\langle 3\rangle\dfrac{\langle 11\rangle\dfrac{}{\sigma_{1\doteq 3}(\cdots)\vdash\{1\}\multimap 3}\qquad \langle 13\rangle\dfrac{\langle 3\rangle\dfrac{\langle 13\rangle\dfrac{\langle 9\rangle\dfrac{E_2\vdash\{1\}\multimap 2}{E_1\times E_2\vdash\{3\}\multimap 4}}{\sigma_{2\doteq 4}(E_1\times E_2)\vdash\{3\}\multimap 4}\qquad\langle 12\rangle\dfrac{}{\sigma_{2\doteq 4}(E_1\times E_2)\vdash\{4\}\multimap 2}}{\sigma_{2\doteq 4}(E_1\times E_2)\vdash\{3\}\multimap 2}}{\sigma_{1\doteq 3}(\sigma_{2\doteq 4}(E_1\times E_2))\vdash\{3\}\multimap 2}}{\sigma_{1\doteq 3}(\sigma_{2\doteq 4}(E_1\times E_2))\vdash\{1\}\multimap 2}}{\pi_{1,2}(\sigma_{1\doteq 3}(\sigma_{2\doteq 4}(E_1\times E_2)))\vdash\{1\}\multimap 2}$$

Note that it is not surprising that different statements are derivable depending on how an operator is expressed. Indeed, equivalence of RA expressions is an undecidable semantic notion, while statement derivability is a decidable syntactic notion.

3.2 Soundness and Completeness of $\mathsf{IS}_{\mathsf{RA}}$

Unsurprisingly, the proposed inference system is sound.

▶ **Theorem 20** (Soundness of $\mathsf{IS}_{\mathsf{RA}}$). *Let E be an $\mathcal{E}$-expression over $\mathcal{S}$.*

1. *If the statement $E\Downarrow$ is derivable in $\mathsf{IS}_{\mathsf{RA}}$, then E is testable.*
2. *If, for some $i \in \mathsf{pos}(E)$ and some $P \subseteq \mathsf{pos}(E)$, the statement $E \vdash P \multimap i$ is derivable in $\mathsf{IS}_{\mathsf{RA}}$, then position i is effectively safe in E relative to P.*

Less typically, our inference system is *effective*, meaning that it is decidable whether a given statement is derivable and if so, a derivation can be computed.

▶ **Proposition 21.** $\mathsf{IS}_{\mathsf{RA}}$ *is effective.*

As our inference system is both sound (Theorem 20) and effective (Proposition 21), it cannot be complete, because both testability and effective safety for RA are undecidable (Proposition 18). From a practical view-point, an advantage of a decidable but incomplete inference system over an undecidable but complete one, is that it defines a decidable class of safe and computable queries. When such a system is integrated into a query engine, the user can get transparent guarantees, rather than best-effort behaviour based on opaque heuristics: the engine can show the rules to the user and promise to derive everything derivable.

While $\mathsf{IS}_{\mathsf{RA}}$ is incomplete in general, it does offer some *relative* completeness. First, as a basic sanity check, one may verify that $\mathsf{IS}_{\mathsf{RA}}$ is complete for RA expressions without external predicates. As these are all testable and effectively safe (Prop. 18), we have the following.

▶ **Proposition 22.** *For every RA expression E without external predicates, $E\Downarrow$ and $E \vdash \varnothing \multimap i$ are derivable in* $\mathsf{IS}_{\mathsf{RA}}$ *for every $i \in \mathsf{pos}(E)$.*

Moreover, we can prove that $\mathsf{IS}_{\mathsf{RA}}$ is complete *up to query equivalence* when only trivial access patterns are used in the access schema.

▸ **Theorem 23.** *Assume that $\mathcal{E}(L) = \{[\mathsf{ar}(L)] \rightsquigarrow \varnothing\}$ for all $L \in \mathcal{E}$. Let E be a uniformly computable $\mathcal{E}$-expression over $\mathcal{S}$. Then there is an $\mathcal{E}$-expression $\hat{E}$ such that $[\![\hat{E}]\!] = [\![E]\!]$, and statements $\hat{E}\Downarrow$ and $\hat{E} \vdash \varnothing \multimap i$ are derivable in $\mathsf{IS}_{\mathsf{RA}}$ for all positions i of $\hat{E}$.*

4 First-Order Logic with External Predicates

We now move to first-order logic (FO). Due to the presence of external predicates, we work with the classical semantics, in which quantified variables range over the infinite set of values $\mathbb{V}$, rather than the active domain semantics. Note that we cannot reason about safety of FO formulas by translating FO to RA: we know that *safe* FO equals RA, but this says nothing about unsafe FO formulas. We need to know which formulas are safe before we can translate them into RA. Could we have started with FO instead and handled RA by translation to FO? Such a translation is possible in principle, but since no inference system for safety can be complete, the resulting notion of safety for RA would then be translation-dependent. In practice, this would mean that a user writing a query would be getting errors about the translated query, not the one originally formulated.

We define syntax and semantics of first-order logic adjusted to handle external predicates as follows. Let $\mathbb{X}$ be a set of variables, $\mathcal{S}$ a database schema, and $\mathcal{E}$ an access schema. The syntax of *formulas φ over an access schema $\mathcal{E}$ and database schema $\mathcal{S}$*, also referred to as *$\mathcal{E}$-formulas*, and *terms t* is given by the grammar

$$\varphi ::= R(\overline{x}) \mid L(\overline{x}) \mid t = t \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x\, \varphi \mid \forall x\, \varphi \qquad\qquad t ::= c \mid x$$

where $R \in \mathcal{S}$, $L \in \mathcal{E}$, $c \in \mathbb{V}$, $x \in \mathbb{X}$ and $\overline{x} \in \mathbb{X}^*$. For simplicity, we assume that if we have $\exists x\ \varphi$ or $\forall x\ \varphi$, then x appears in φ. We use the classical notion of free variables of a formula φ, denoted by $\mathsf{fv}(\varphi)$. The semantics of a formula φ is defined with respect to an $\mathcal{E}$-interpretation λ, a database D over $\mathcal{S}$, and a valuation $\mu : \mathsf{fv}(\varphi) \to \mathbb{V}$ of the free variables of φ:

- $\lambda, D, \mu \models R(\overline{x})$ iff $\mu(\overline{x}) \in D(R)$ and $\lambda, D, \mu \models L(\overline{x})$ iff $\mu(\overline{x}) \in \lambda(L)$;
- the interpretation of Boolean connectives $\wedge$, $\vee$ and $\neg$ is standard;
- $\lambda, D, \mu \models \exists x\ \varphi$ iff there is $v \in \mathbb{V}$ such that $\lambda, D, \mu[x\mapsto v] \models \varphi$ where the valuation $\mu[x\mapsto v]$ extends μ with $\mu[x\mapsto v](x) = v$;
- $\lambda, D, \mu \models \forall x\ \varphi$ iff $\lambda, D, \mu[x\mapsto v] \models \varphi$ for every $v \in \mathbb{V}$.

For an $\mathcal{E}$-formula φ over $\mathcal{S}$, we let $[\![\varphi]\!]\,(\lambda, D) = \{\mu : \mathsf{fv}(\varphi) \to \mathbb{V} \mid \lambda, D, \mu \models \varphi\}$ for all $\lambda \in \Lambda(\mathcal{E})$ and $D \in \mathrm{DB}(\mathcal{S})$. Recall that an $\mathsf{fv}(\varphi)$-tuple is a function $\mathsf{fv}(\varphi) \to \mathbb{V}$, hence $[\![\varphi]\!]$ is an $\mathcal{E}$-query over database schema $\mathcal{S}$ and output schema $\mathsf{fv}(\varphi)$. We call formula φ *testable*, *effectively safe (relative to a set $X \subseteq \mathsf{fv}(\varphi)$ of variables)*, or *uniformly computable*, if so is query $[\![\varphi]\!]$. Similarly, a variable $y \in \mathsf{fv}(\varphi)$ is *effectively safe in φ (relative to $X \subseteq \mathsf{fv}(\varphi)$)*, if so is field y in the query $[\![\varphi]\!]$.

Also, to handle unrestricted negation $\neg\varphi$ (the difference operator $E_1 - E_2$ in RA is *guarded* negation), we introduce a dual notion of *co-safety*; it will facilitate meaningful inferences for formulas outside the positive fragment of FO. Variable $y \in \mathsf{fv}(\varphi)$ is *effectively co-safe in φ (relative to $X \subseteq \mathsf{fv}(\varphi)$)* if y is effectively safe in $\neg\varphi$ (relative to X).

By the correspondence between RA and FO, we have an analogue of Proposition 18 for FO. In particular, testability and effective (co-)safety are undecidable for FO formulas.

4.1 Inference System $\mathsf{IS}_{\mathsf{FO}}$

Like for RA, we use a sequent-like notation. For $X \subseteq \mathsf{fv}(\varphi)$ and $y \in \mathsf{fv}(\varphi)$, we write $\varphi \vdash X \multimap y$ for "y is effectively safe in φ relative to X", and $\varphi \vdash X \multimap\!\!\bullet\, y$ for "y is effectively co-safe in φ

relative to X". Moreover, we represent rules that are identical for safety and co-safety as a single rule, with the symbol $\mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}}$ representing either $\multimap$ or $\multimapdot$. Note that this choice must be consistent when a rule is applied: if $\mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}}$ appears multiple times in a rule, then either all occurrences are interpreted as $\multimap$ or all are interpreted as $\multimapdot$.

Axiom, weakening, and cut

$$\langle 24\rangle\frac{}{\varphi \vdash \{x\} \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} x} \qquad \langle 25\rangle\frac{\varphi \vdash X \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} y}{\varphi \vdash X \cup \{z\} \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} y} \qquad \langle 26\rangle\frac{\varphi \vdash X \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} y \quad \varphi \vdash X' \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} y'}{\varphi \vdash X \cup (X' - \{y\}) \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} y'}$$

Atoms

$$\langle 27\rangle\frac{}{y = v \vdash \varnothing \multimap y}\, y \in \mathbb{X}, v \in \mathbb{V} \qquad \langle 28\rangle\frac{}{v = y \vdash \varnothing \multimap y}\, y \in \mathbb{X}, v \in \mathbb{V} \qquad \langle 29\rangle\frac{}{y = x \vdash \{x\} \multimap y}\, x, y \in \mathbb{X}$$

$$\langle 30\rangle\frac{}{x = y \vdash \{x\} \multimap y}\, x, y \in \mathbb{X} \qquad \langle 31\rangle\frac{}{R(x_1, \ldots, x_n) \vdash \varnothing \multimap x_j}\, R \in \mathcal{S}, j \in \{1, \ldots, n\}$$

$$\langle 32\rangle\frac{}{L(x_1, \ldots, x_n) \vdash \{x_i \mid i \in I\} \multimap x_j}\, L \in \mathcal{E},\ I \rightsquigarrow O \in \mathcal{E}(L),\ j \in O$$

Boolean operations

$$\langle 33\rangle\frac{\varphi_1 \vdash X_1 \multimap y \qquad \varphi_2 \vdash X_2 \multimap y}{\varphi_1 \vee \varphi_2 \vdash X_1 \cup X_2 \multimap y} \qquad \langle 34\rangle\frac{\varphi_1 \vdash X_1 \multimapdot y \qquad \varphi_2 \vdash X_2 \multimapdot y}{\varphi_1 \wedge \varphi_2 \vdash X_1 \cup X_2 \multimapdot y}$$

$$\langle 35\rangle\frac{\varphi_1 \vdash X_1 \multimap y}{\varphi_1 \wedge \varphi_2 \vdash X_1 \multimap y} \qquad \langle 36\rangle\frac{\varphi_1 \vdash X_2 \multimapdot y}{\varphi_1 \vee \varphi_2 \vdash X_2 \multimapdot y} \qquad \langle 37\rangle\frac{\varphi \vdash X \multimapdot y}{\neg\varphi \vdash X \multimap y}$$

$$\langle 38\rangle\frac{\varphi_2 \vdash X_2 \multimap y}{\varphi_1 \wedge \varphi_2 \vdash X_2 \multimap y} \qquad \langle 39\rangle\frac{\varphi_2 \vdash X_2 \multimapdot y}{\varphi_1 \vee \varphi_2 \vdash X_2 \multimapdot y} \qquad \langle 40\rangle\frac{\varphi \vdash X \multimap y}{\neg\varphi \vdash X \multimapdot y}$$

Quantifiers

$$\langle 41\rangle\frac{\varphi \vdash X \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} y}{\exists z\, \varphi \vdash X \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} y}\, z \neq y, z \notin X \qquad \langle 42\rangle\frac{\varphi \vdash X \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} y}{\forall z\, \varphi \vdash X \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} y}\, z \neq y, z \notin X$$

Testability

$$\langle 43\rangle\frac{}{t_1 = t_2 \Downarrow}\, t_1, t_2 \in \mathbb{V} \cup \mathbb{X} \qquad \langle 44\rangle\frac{}{R(x_1, \ldots, x_n) \Downarrow}\, R \in \mathcal{S} \qquad \langle 45\rangle\frac{}{L(x_1, \ldots, x_n) \Downarrow}\, L \in \mathcal{E}$$

$$\langle 46\rangle\frac{\varphi_1 \Downarrow \qquad \varphi_2 \Downarrow}{(\varphi_1\, \theta\, \varphi_2) \Downarrow}\, \theta \in \{\vee, \wedge\} \qquad \langle 47\rangle\frac{\varphi \Downarrow}{(\neg\varphi) \Downarrow}$$

Testability of quantified formulas requires some minimal safety of the eliminated variable.

$$\langle 48\rangle\frac{\varphi \Downarrow \qquad \varphi \vdash \mathsf{fv}(\varphi) - \{z\} \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} z}{(\exists z\, \varphi) \Downarrow}\, z \in \mathsf{fv}(\varphi) \qquad \langle 49\rangle\frac{\varphi \Downarrow \qquad \varphi \vdash \mathsf{fv}(\varphi) - \{z\} \mathrel{\underset{\bullet}{\overset{\circ}{\rightrightarrows}}} z}{(\forall z\, \varphi) \Downarrow}\, z \in \mathsf{fv}(\varphi)$$

4.2 Soundness and Completeness

As in the case of RA, our inference system for FO is sound and effective.

▸ **Theorem 24** (Soundness of $\mathsf{IS}_{\mathsf{FO}}$). *Let φ be an $\mathcal{E}$-formula.*

1. *If the statement $\varphi\Downarrow$ is derivable in $\mathsf{IS}_{\mathsf{FO}}$, then φ is testable.*
2. *For $X \subseteq \mathsf{fv}(\varphi)$ and $y \in \mathsf{fv}(\varphi)$, if the statement $\varphi \vdash X \multimap y$ (resp. $\varphi \vdash X \multimapdot y$) is derivable in $\mathsf{IS}_{\mathsf{FO}}$, then variable y is effectvely safe (resp. effectively co-safe) in φ relative to X.*

▸ **Proposition 25.** $\mathsf{IS}_{\mathsf{FO}}$ *is effective.*

As for RA, this precludes general completeness, but we can show that $\mathsf{IS}_{\mathsf{FO}}$ is complete for *safe-range FO formulas* [1, Chapter 5] (see the appendix for the definition). The importance of safe-range FO formulas stems from the following folklore result concerning safe queries (i.e. that always return a finite set of tuples, as per Definition 11).

▸ **Theorem 26** (Folklore). *For every FO formula φ without external predicates such that the query $[\![\varphi]\!]$ is safe, there is a safe-range FO formula φ' without external predicates such that $[\![\varphi]\!] = [\![\varphi']\!]$.*

Note that in the absence of external predicates, all safe FO formulas are effectively safe (because they are testable). The class of effectively safe formulas defined by $\mathsf{IS}_{\mathsf{FO}}$ includes all safe-range FO formulas.

▸ **Theorem 27.** *Let φ be an FO formula without external predicates that is safe-range. Then, the statements $\varphi\Downarrow$ and $\varphi \vdash \varnothing \multimap x$ for each $x \in \mathsf{fv}(\varphi)$ can be derived in $\mathsf{IS}_{\mathsf{FO}}$.*

Note that our system avoids normalizing formulas by relying on co-safety. Let us see that $\mathsf{IS}_{\mathsf{FO}}$ can derive effective safety also for some non-safe-range formulas without external predicates.

▸ **Example 28.** Consider the formula $\varphi(y) := \neg\exists x\,(\neg R(x,y) \vee S(x))$ with $R, S \in \mathcal{S}$. Note that $\varphi(y)$ is trivially false for each y because R is finite. Using our inference system one can derive $\varphi \vdash \varnothing \multimap y$ by alternating between safety and co-safety: starting from $R(x,y) \vdash \varnothing \multimap y$, we can apply rules ⟨40⟩, ⟨36⟩, ⟨41⟩, and ⟨40⟩ to derive $\neg\exists x\;(\neg R(x,y) \vee S(x)) \vdash \varnothing \multimap y$. However, φ is not safe-range.

Analogously to the RA case, we are now ready to prove completeness of the inference system of Section 4.1, up to query equivalence, in the case of trivial access patterns.

▸ **Theorem 29.** *Assume that $\mathcal{E}(L) = \{[\mathsf{ar}(L)] \rightsquigarrow \varnothing\}$ for all $L \in \mathcal{E}$. Let φ be a uniformly computable $\mathcal{E}$-formula over $\mathcal{S}$. Then there is a $\mathcal{E}$-formula $\hat{\varphi}$ such that $[\![\hat{\varphi}]\!] = [\![\varphi]\!]$, and statements $\hat{\varphi}\Downarrow$ and $\hat{\varphi} \vdash \varnothing \multimap x$ are derivable in $\mathsf{IS}_{\mathsf{FO}}$ for all $x \in \mathsf{fv}(\hat{\varphi})$.*

Note that this result does not contradict [30] which states that safe queries may not have effective syntax for *a particular* interpretation λ, as queries safe under that interpretation need not be safe under all interpretations (and thus they need not be uniformly computable).

5 Combining Algebra and Logic

We are now ready to approach the combination of relational algebra and first-order logic, announced in the introduction. All we need to do is connect RA expressions and FO formulas. This is done by turning each formula into an algebra expression that produces the set of valuations of free variables that satisfy the formula (*comprehension*), and turning each algebra expression of arity n into a formula with n free variables (*membership*).

5.1 Syntax and Semantics

The grammars for $\mathcal{E}$-expressions over $\mathcal{S}$ and $\mathcal{E}$-formulas over $\mathcal{S}$ are combined into one and extended with productions

$$E \;::=\; \{\overline{x} : \varphi\} \quad \textit{(Comprehension)} \qquad \text{and} \qquad \varphi \;::=\; E(\overline{x}) \quad \textit{(Membership)}$$

where $\overline{x} \in \mathbb{X}^*$. We refer to the resulting expressions as *FO/RA expressions.* Observe that an expression E can now have free variables, denoted by $\mathsf{fv}(E)$. We let

$$\mathsf{fv}(\{x_1,\ldots,x_k : \varphi\}) = \mathsf{fv}(\varphi) - \{x_1,\ldots,x_k\} \quad \text{and} \quad \mathsf{fv}\big(E(x_1,\ldots,x_k)\big) = \mathsf{fv}(E) \cup \{x_1,\ldots,x_k\}\,.$$

Note also that the productions $\varphi ::= R(\overline{x})$ and $\varphi ::= L(\overline{x})$ are now redundant because they are generalized by the membership rule above, and can therefore be omitted.

The semantics of both formulas and expressions is given with respect to a database D, an interpretation λ of external predicates, and a valuation μ of variables (which can now occur also in expressions). We define the semantics of comprehension and membership (which make formulas and expressions interdependent) as follows:

$$[\![\{x_1,\dots,x_k : \varphi\}]\!]_\mu(\lambda, D) = \{(v_1,\dots,v_k) \in \mathbb{V}^k \mid \lambda, D, \mu[x_1 \mapsto v_1,\dots,x_k \mapsto v_k] \models \varphi\}$$
$$\lambda, D, \mu \models E(x_1,\dots,x_k) \quad \text{iff} \quad \big(\mu(x_1),\dots,\mu(x_k)\big) \in [\![E]\!]_\mu(\lambda, D)$$

A FO/RA formula φ defines a query $[\![\varphi]\!]$ over output schema $\mathsf{fv}(\varphi)$,

$$[\![\varphi]\!](\lambda, D) = \big\{\mu \in \mathbb{V}^{\mathsf{fv}(\varphi)} \;\big|\; \lambda, D, \mu \models \varphi\big\}.$$

A FO/RA expression E defines a query $[\![E]\!]$ over output schema $\mathsf{pos}(E) \cup \mathsf{fv}(E)$,

$$[\![E]\!](\lambda, D) = \big\{\alpha \cup \mu \;\big|\; \alpha \in \mathbb{V}^{\mathsf{pos}(E)},\ \mu \in \mathbb{V}^{\mathsf{fv}(E)},\ \alpha \in [\![E]\!]_\mu(\lambda, D)\big\}.$$

▸ **Example 30.** Let R and S be binary predicates. The expression $\{(x,y) : \exists z\ R(x,z) \wedge R(z,y)\}$ returns endpoints of paths of length two in relation R, while $\{(x,y) : \exists z\ (R-S)(x,z) \wedge (R-S)(z,y)\}$ returns endpoints of paths of length two in $R - S$.

The definitions of uniform computability, testability, as well as (relative) effective safety and co-safety carry over immediately to FO/RA formulas and expressions. In particular, $o \in \mathsf{pos}(E) \cup \mathsf{fv}(E)$ *is effectively safe in* E *relative to* $\Delta \subseteq \mathsf{fv}(E) \cup \mathsf{pos}(E)$ if so is o in $[\![E]\!]$.

5.2 Inference System $\mathsf{IS}_{\mathsf{FO/RA}}$

The inference system for FO/RA combines the systems for FO and RA, along with additional rules that allow moving between formulas and expressions. As before, we use a sequent-like notation: for $\Delta \subseteq \mathsf{pos}(E) \cup \mathsf{fv}(E)$ and $o \in \mathsf{pos}(E) \cup \mathsf{fv}(E)$, we write $E \vdash \Delta \multimap o$ for "o is effectively safe in E relative to Δ".

$\mathsf{IS}_{\mathsf{FO/RA}}$ consists of three parts. The first part is $\mathsf{IS}_{\mathsf{FO}}$, as defined in Section 4.1. The second part is $\mathsf{IS}_{\mathsf{RA}}$ from Section 3.1, slightly generalized to accommodate both positions and variables in safety rules. More precisely, in axiom, weakening, and cut, we simply replace P, P' and i, i' with Δ, Δ' and o, o'. Rules for atomic expressions remain the same. In rules for RA operations, we replace P, P_1, P_2 and i, i' with Δ, Δ_1, Δ_2 and o, o' in rules ⟨7⟩, ⟨8⟩, ⟨10⟩, and ⟨13⟩. Rules ⟨11⟩, ⟨12⟩, and ⟨14⟩ are not changed. Rule ⟨9⟩ is replaced with rules

$$\langle 50\rangle\frac{E_2 \vdash P_2 \cup X_2 \multimap y}{E_1 \times E_2 \vdash (P_2 + \mathsf{ar}(E_1)) \cup X_2 \multimap y} \qquad \langle 51\rangle\frac{E_2 \vdash P_2 \cup X_2 \multimap i}{E_1 \times E_2 \vdash (P_2 + \mathsf{ar}(E_1)) \cup X_2 \multimap i + \mathsf{ar}(E_1)}$$

where $P_2 \subseteq \mathsf{pos}(E_2)$, $X_2 \subseteq \mathsf{fv}(E_2)$, $i \in \mathsf{pos}(E_2)$, and $y \in \mathsf{fv}(E_2)$. Rule ⟨15⟩ is replaced with

$$\langle 52\rangle\frac{E \vdash P \cup X \multimap i}{\pi_s(E) \vdash P' \cup X \multimap i'}\, s(P') = P, s(i') = i \qquad \langle 53\rangle\frac{E \vdash P \cup X \multimap y}{\pi_s(E) \vdash P' \cup X \multimap y}\, s(P') = P$$

where $P \subseteq \mathsf{pos}(E)$, $X \subseteq \mathsf{fv}(E)$, $i \in \mathsf{pos}(E)$, $P' \subseteq \mathsf{pos}(\pi_s(E))$, $i' \in \mathsf{pos}(\pi_s(E))$, and $y \in \mathsf{fv}(E)$. Rules for testability are not changed except that $\mathsf{set}(s)$ in rule ⟨21⟩, is replaced by $\mathsf{set}(s) \cup \mathsf{fv}(E)$.

The third part of $\mathsf{IS}_{\mathsf{FO/RA}}$ takes care of the two new productions that turn formulas into expressions and the other way around. For a set Δ, we let $\Delta[o/o']$ denote Δ itself when $o \notin \Delta$, and $(\Delta - \{o\}) \cup \{o'\}$ otherwise. Moreover, we write $\Delta[o_1/o_1', \dots, o_k/o_k']$, where all o_i's are different, as a shorthand for $\Delta[o_1/o_1'] \cdots [o_k/o_k']$. With this in place, we introduce the following safety rules for comprehension and membership productions.

$$\langle 54\rangle\frac{E \vdash \Delta \multimap i}{E(x_1,\dots,x_n) \vdash \Delta[1/x_1,\dots,n/x_n] \multimap x_i} \qquad \langle 55\rangle\frac{E \vdash \Delta \multimap y}{E(x_1,\dots,x_n) \vdash \Delta[1/x_1,\dots,n/x_n] \multimap y}$$

$$\langle 56\rangle\frac{\varphi \vdash \Delta \multimap x_i}{\{x_1,\dots,x_n : \varphi\} \vdash \Delta[x_1/1,\dots,x_n/n] \multimap i} \qquad \langle 57\rangle\frac{\varphi \vdash \Delta \multimap y}{\{x_1,\dots,x_n : \varphi\} \vdash \Delta[x_1/1,\dots,x_n/n] \multimap y}$$

$$\langle 58\rangle\frac{}{\{x_1,\ldots,x_n:\varphi\}\vdash\{i\}\multimap j}\,x_i=x_j\qquad\langle 59\rangle\frac{\varphi\Downarrow}{\{\overline{x}:\varphi\}\Downarrow}\,\overline{x}\in\mathbb{X}^*\qquad\langle 60\rangle\frac{E\Downarrow}{E(\overline{x})\Downarrow}\,\overline{x}\in\mathbb{X}^*$$

Note that going from expressions to formulas and back is transparent for testability, as shown by rules ⟨59⟩ and ⟨60⟩ above.

Finally, we show that $\mathsf{IS}_{\mathsf{FO/RA}}$ is sound (Theorem 31), and effective (Proposition 32).

Theorem 31 (Soundness of $\mathsf{IS}_{\mathsf{FO/RA}}$). *Let ξ be an $\mathcal{E}$-expression or an $\mathcal{E}$-formula.*

1. *If the statement $\xi\Downarrow$ is derivable in the $\mathsf{IS}_{\mathsf{FO/RA}}$, then ξ is testable.*
2. *For $\Delta\subseteq\mathsf{pos}(\xi)\cup\mathsf{fv}(\xi)$ and $o\in\mathsf{pos}(\xi)\cup\mathsf{fv}(\xi)$, if $\xi\vdash\Delta\multimap o$ (resp. $\xi\vdash\Delta\multimap\!\!\bullet\, f$) is derivable in $\mathsf{IS}_{\mathsf{FO/RA}}$, then o is effectively safe (resp. co-safe) in ξ relative to Δ.*

Proposition 32. *$\mathsf{IS}_{\mathsf{FO/RA}}$ is effective.*

6 Related Work

Our work is closely related to the notion of access patterns [23]: those state that values of some attributes must be given to retrieve tuples from a relation. While our access patterns are similar to those studied in the literature [12, 14, 22, 26, 13] they are more general since we allow patterns of the form $I\rightsquigarrow O$ where $I\cup O$ is not the entire set of attributes. Our notion of a set X *supporting access pattern* $I\rightsquigarrow O$ is similar to having a *finiteness constraint* $I\rightsquigarrow O$ on X [29] but is again more general. For instance even if for all positions I in X, the number of values for O is finite (that is, X satisfies finiteness constraint $I\rightsquigarrow O$ in [29]), this does not mean that $I\rightsquigarrow O$ needs to allow the access pattern $I\rightsquigarrow O$. For example, even though the set of phone numbers is finite, an external predicate (or web service) may only allow access to a phone number if a correct name and address is provided.

Furthermore, in the literature on access patterns, relations are finite, and most of that work focus on conjunctive queries, their unions, and slight extensions [26, 20], though some of it was extended to the entire relational calculus [25]. The latter served as the basis for deriving effective computability in a related setting of bounded evaluation [13, 9]. That work itself came from formalizing the notion of scale independence in big data querying [3].

Both the access pattern framework [26] and the bounded evaluation framework [13] present (essentially) derivation rules for first-order logic queries, that are similar to our rules for safety in Section 4. These however do not touch infinite relations, nor co-safety, nor testability, and only work in the context of FO (no RA or FO/RA). Importantly, the notions of computability targeted in these works are rather specific and vary from one framework to another (e.g., annotated query plans [14], stability [22], $\mathcal{V}$-executability [26]). Here, we study a general notion of computability in the presence of access patterns (based on Turing machines with oracles) that subsumes previously considered notions while remaining close to execution models for queries with UDFs.

Safety with infinite predicates was studied in the settings of constraint databases and Datalog queries. Regarding the former [21], we know that in general an analog of SafeFO defined via range-restriction is impossible: there are computable external predicates for which the safe queries cannot be captured by a recursively enumerable class of FO queries [30]. A known case of safety captured by range-restriction applies to external predicates defined over $\mathbb{R}$ with the usual arithmetic operations and comparisons [8]. However, these results already break over $\mathbb{Q}$ or $\mathbb{Z}$, which makes them mostly of theoretical interest. Safety of positive Datalog programs in the presence of infinite predicates is discussed in [27, 29, 11]. In that case, safety breaks down into two components: safety of one step of the recursion (called *weak safety*) and termination of the recursion. Most of the work is about the termination of the program. It is undecidable in general [29], and several conditions on the IDBs are known to make it decidable [29, 11]. Weak safety of positive Datalog programs corresponds in our setting to safety of UCQs. It is shown to be EXPTIME-complete in [29] and another algorithm is given in [11]. In both cases, propagation of finiteness dependencies appears in proofs, but no inference system is explicitly formalised.

Computability in the presence of infinite relations can be quite intricate. Classically queries are defined as computable maps from finite databases to finite relations [10, 2]. When the finiteness assumption is relaxed, several approaches exist. In recursive databases [17, 18], relations are possibly infinite but computable, and are given by their defining Turing machines. This model finds its origins in the theory of computability over recursive graphs [7, 6]. However in general such infinite computable relations are not even closed under FO operations, unless they are highly symmetric. Another related setting is that of metafinite model theory [15], in which database elements can be associated with elements of an infinite structure (such as, for example, real numbers with arithmetic). While a number of expressiveness results were obtained in [15], the development of a computational model for this setting was left for future work, but it was never completed to the best of our knowledge. On the other hand, a sufficient condition for a positive Datalog program to be computable is given in [11]: in each rule, the body variables need to be safe relative to the head variables. It is related to our rule $\langle 48 \rangle$, and their condition would be that this rule is applicable for every existential quantifier.

7 Conclusion

We formalized queries with external predicates along with a corresponding notion of computability, requiring the existence of a *uniform* algorithm that communicates with an oracle for each access pattern to an external predicate. This means that the same algorithm (where only oracles change) always computes the correct query result and works for every possible interpretation of the external predicates. We also characterized computability of arbitrary queries in terms of their effective safety and testability.

In our quest for computability, we presented effective rule systems for inferring effective safety and testability for RA, FO, and FO/RA. Our inference systems can used as working components of an actual DBMS, providing a transparent and effective sufficient criterion for computability. When such an inference system is integrated in a query engine, the user gets transparent guarantees, rather than best-effort behavior based on internal compiler heuristics. The latter happens in languages such as Rel and .QL, where safety guarantees are inferred from the compiler's behavior (that may well change) rather than cast in stone in the documentation. We make another important step towards applicability in real-life languages by studying safety and computability in a language that combines FO with RA, as happens in many relational query languages. We believe a variety of language-specific features can be covered simply by including additional rules into the inference system. An interesting question is how to use the insights gained during inference to build query plans.

A fundamental open question is the completeness of our inference systems. Safety, effective safety, and testability are undecidable for RA and FO, so any effective inference system for these notions must be incomplete, but we did prove relative completeness (up to query equivalence) in the case when the external schema only provides trivial access patterns. Can this be extended to arbitrary access patterns? A positive answer would mean that derivable computability, despite being more restrictive than computability, does not additionally limit the expressive power of queries.

References

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

2 Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. *Database Theory*. Open source at `https://github.com/pdm-book/community`, 2022.

3 Michael Armbrust, Eric Liang, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. Generalized scale independence through incremental precomputation. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 625–636. ACM, 2013. `doi:10.1145/2463676.2465333`.

4 S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2006. URL: `https://theory.cs.princeton.edu/complexity/book.pdf`.

5 Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. URL: `https://doi.org/10.4230/LIPIcs.ECOOP.2016.2`, `doi:10.4230/LIPICS.ECOOP.2016.2`.

6 D. R. Bean. Recursive Euler and Hamilton paths. *Proc. Amer. Math. Soc.*, 55:385–394, 1976.

7 Richard Beigel and William I. Gasarch. On the complexity of finding the chromatic number of a recursive graph I: the bounded case. *Ann. Pure Appl. Log.*, 45(1):1–38, 1989. `doi:10.1016/0168-0072(89)90029-8`.

8 Michael Benedikt and Leonid Libkin. Safe constraint queries. *SIAM J. Comput.*, 29(5):1652–1682, 2000. `doi:10.1137/S0097539798342484`.

9 Yang Cao, Wenfei Fan, and Tengfei Yuan. Bounded evaluation: Querying big data with bounded resources. *Int. J. Autom. Comput.*, 17(4):502–526, 2020. URL: `https://doi.org/10.1007/s11633-020-1236-1`, `doi:10.1007/S11633-020-1236-1`.

10 Ashok K. Chandra and David Harel. Computable queries for relational data bases. *J. Comput. Syst. Sci.*, 21(2):156–178, 1980. `doi:10.1016/0022-0000(80)90032-X`.

11 Sara Cohen, Joseph (Yossi) Gil, and Evelina Zarivach. Datalog programs over infinite databases, revisited. In Marcelo Arenas and Michael I. Schwartzbach, editors, *Database Programming Languages*, pages 32–47, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

12 Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting queries using views with access patterns under integrity constraints. *Theor. Comput. Sci.*, 371(3):200–226, 2007. URL: `https://doi.org/10.1016/j.tcs.2006.11.008`, `doi:10.1016/J.TCS.2006.11.008`.

13 Wenfei Fan, Floris Geerts, and Leonid Libkin. On scale independence for querying big data. In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 51–62. ACM, 2014. `doi:10.1145/2594538.2594551`.

14 Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 311–322. ACM Press, 1999. `doi:10.1145/304182.304210`.

15 Erich Grädel and Yuri Gurevich. Metafinite model theory. *Inf. Comput.*, 140(1):26–81, 1998. URL: `https://doi.org/10.1006/inco.1997.2675`, `doi:10.1006/INCO.1997.2675`.

16 Erich Grädel, Phokion G. Kolaitis, Leonid Libkin, Maarten Marx, Joel Spencer, Moshe Y. Vardi, Yde Venema, and Scott Weinstein. *Finite Model Theory and Its Applications*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2007. `doi:10.1007/3-540-68804-8`.

17 Tirza Hirst and David Harel. Completeness results for recursive data bases. In Catriel Beeri, editor, *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles*

of Database Systems, May 25-28, 1993, Washington, DC, USA, pages 244–252. ACM Press, 1993. doi:10.1145/153850.153905.

18 Tirza Hirst and David Harel. More about recursive structures: Descriptive complexity and zero-one laws. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 334–347. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561361.

19 Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 422–430. Springer, 2016. doi:10.1007/978-3-319-41540-6_23.

20 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Conjunctive queries with free access patterns under updates. In Floris Geerts and Brecht Vandevoort, editors, *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*, volume 255 of *LIPIcs*, pages 17:1–17:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ICDT.2023.17, doi:10.4230/LIPICS.ICDT.2023.17.

21 Gabriel Kuper, Leonid Libkin, and Jan Paredaens. *Constraint Databases.* Springer, 2000.

22 Chen Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB J.*, 12(3):211–227, 2003. URL: https://doi.org/10.1007/s00778-002-0085-6, doi:10.1007/S00778-002-0085-6.

23 Chen Li and Edward Y. Chang. On answering queries in the presence of limited access patterns. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2001. doi:10.1007/3-540-44503-X_15.

24 Leonid Libkin. *Elements of Finite Model Theory.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. URL: http://www.cs.toronto.edu/%7Elibkin/fmt.

25 Alan Nash and Bertram Ludäscher. Processing first-order queries under limited access patterns. In Catriel Beeri and Alin Deutsch, editors, *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 307–318. ACM, 2004. doi:10.1145/1055558.1055601.

26 Alan Nash and Bertram Ludäscher. Processing unions of conjunctive queries with negation under limited access patterns. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, volume 2992 of *Lecture Notes in Computer Science*, pages 422–440. Springer, 2004. doi:10.1007/978-3-540-24741-8_25.

27 R. Ramakrishnan, F. Bancilhon, and A. Silberschatz. Safety of recursive horn clauses with infinite relations. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '87, page 328–339, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/28659.28694.

28 RelationalAI, 2024. https://learn.relational.ai/.

29 Y. Sagiv and M. Y. Vardi. Safety of datalog queries over infinite databases. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, page 160–171, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/73721.73738.

30 Alexei P. Stolboushkin and Michael A. Taitslin. Finite queries do not have effective syntax. *Inf. Comput.*, 153(1):99–116, 1999. URL: https://doi.org/10.1006/inco.1999.2792, doi:10.1006/INCO.1999.2792.

A Supplementary Materials for Section 2 (Framework)

▸ **Proposition 10.** *Let q be a uniformly computable $\mathcal{E}$-query over database schema $\mathcal{S}$ and output schema $\mathcal{F}$. For every database D over $\mathcal{S}$ and every $\mathcal{E}$-interpretation λ there is a finite $\mathcal{E}$-interpretation λ' such that $\lambda'(L) \subseteq \lambda(L)$ for all $L \in \mathcal{E}$, and $q(\lambda, D) = q(\lambda', D)$. Moreover, some such λ' is uniformly computable from D.*

Proof. Let $\mathcal{A}$ be an $\mathcal{E}$-algorithm that computes q. Let D be a database over $\mathcal{S}$, and λ an $\mathcal{E}$-interpretation. Let us run the algorithm $\mathcal{A}$ on input D with $\mathcal{E}$-interpretation λ, and log all calls to each oracle along with the received answers. For every call to oracle $L_{I \rightsquigarrow O}$ with input $\alpha : I \to \mathbb{V}$ and every tuple $\beta : O \to \mathbb{V}$ received in return, there is a witnessing tuple $\gamma \in \lambda(L)$ that extends both α and β. Let us pick one such γ for each α and β, and let $\widehat{\lambda}(L)$ be the (finite) set of all tuples picked for all oracles $L_{I \rightsquigarrow O}$ with $I \rightsquigarrow O$ ranging over $\mathcal{E}(L)$. Because $\widehat{\lambda}(L)$ is finite, it supports all access patterns in $\mathcal{E}(L)$. Hence, λ is an $\mathcal{E}$-interpretation. We claim that $\mathcal{A}^{\lambda}(D) = \mathcal{A}^{\widehat{\lambda}}(D)$. Indeed, for each logged oracle call, the answer is the same with interpretation λ and with interpretation $\widehat{\lambda}$. A simple argument by induction on the number of steps shows that the runs of $\mathcal{A}$ with interpretation λ and with interpretation $\widehat{\lambda}$ are the same, which gives the claim. Because the algorithm computes q, we have $q(\lambda, D) = \mathcal{A}^{\lambda}(D) = \mathcal{A}^{\widehat{\lambda}}(D) = q(\widehat{\lambda}, D)$.

Every choice of witnessing tuples gives a finite interpretation $\widehat{\lambda}$ such that $q(\lambda, D) = q(\widehat{\lambda}, D)$. Let us see that we can uniformly compute some such $\widehat{\lambda}$ from D. For every call to oracle $L_{I \rightsquigarrow O}$ with input α and every β received in return, rather than guessing a witnessing tuple $\gamma \in \lambda(L)$, we compute it: we iterate over all tuples $\gamma \in \mathbb{V}^{\mathsf{ar}(L)}$ that extend both α and β, and for each such γ call the oracle $L_{[\mathsf{ar}(L)] \rightsquigarrow \varnothing}$. Because some witnessing γ exists, the oracle will give the positive answer eventually. Then we add the current γ to $\widehat{\lambda}(L)$. This gives a terminating $\mathcal{E}$-algorithm that computes some $\widehat{\lambda}$ such that $q(\lambda, D) = q(\widehat{\lambda}, D)$. ◂

▸ **Example 33.** Consider the binary relation H containing all pairs (M, k) where M is a Turing machine that halts in exactly k steps on the empty word. Notice that H supports $\{1, 2\} \rightsquigarrow \varnothing$; indeed, it is decidable to test if a given (M, k) is in H or not. Let Halts be an external predicate symbol and $\mathcal{E}(\mathsf{Halts}) = \{\{1, 2\} \rightsquigarrow \varnothing\}$. Consider now a database relation symbol TM and the database D mapping TM to a finite set of (encodings of) Turing machines we are interested in.

The query returning all x such that $\mathsf{TM}(x) \wedge \exists k\ \mathsf{Halts}(x, k)$ is not only safe, but also effectively safe. Indeed, for every database D and every $\mathcal{E}$-interpretation, the set of returned answers is a subset of D. However, the query is not uniformly computable because any hypothetical $\mathcal{E}$-algorithm $\mathcal{A}$ computing it could be used to solve the halting problem. Indeed, by taking $\lambda : \mathsf{Halts} \mapsto H$, we would have that $\mathcal{A}^{\lambda}(D)$ is the set of all Turing machines in D which halt on the empty word, and this for all D. In particular on databases D_M containing a single Turing machine M, the set $\mathcal{A}^{\lambda}(D_M)$ would be $\{M\}$ iff M halts on the empty word. Notice that $\mathcal{A}^{\lambda}$ is a computable function (in the classical sense). In fact $\mathcal{A}$, under interpretation λ, can be turned into an ordinary algorithm by replacing each call to the Halts oracle by the the decision procedure for H. We would then have an algorithm for the halting problem. For similar reasons the query is not testable (see Definition 14).

▸ **Proposition 15.** *An $\mathcal{E}$-query q is uniformly computable iff it is effectively safe and testable.*

Proof. Let q be an $\mathcal{E}$-query over database schema $\mathcal{S}$ and output schema $\mathcal{F}$.

Suppose that q is uniformly computable. It follows immediately that there is a uniformly computable overapproximation of $q^{\varnothing \rightsquigarrow \mathcal{F}}$: indeed, the query q itself is such overapproximation, modulo interpreting $q : \Lambda(\mathcal{E}) \times \mathrm{DB}(\mathcal{S}) \to \mathbb{V}^{\mathcal{F}}$ as a function $\Lambda(\mathcal{E}) \times \mathrm{DB}(\mathcal{S}) \times \mathbb{V}^{\varnothing} \to \mathbb{V}^{\mathcal{F}}$. Hence, q is effectively safe. Testability also follows easily: the algorithm simply computes the set $q(\lambda, D)$ and checks if it contains the input tuple.

Conversely, suppose that q is effectively safe and testable. It follows that there are two $\mathcal{E}$-algorithms, computing $q^{\mathcal{F} \rightsquigarrow \varnothing}$ and an overapproximation f of $q^{\varnothing \rightsquigarrow \mathcal{F}}$, respectively. An $\mathcal{E}$-algorithm computing q can be obtained as follows. First, run the $\mathcal{E}$-algorithm computing the overapproximation f on the input database D and the $\varnothing$-tuple. Then, for each returned $\mathcal{F}$-tuple β, run the $\mathcal{E}$-algorithm computing $q^{\mathcal{F} \rightsquigarrow \varnothing}$ on D and β, test if it returns $\varnothing$ or the singleton of the $\varnothing$-tuple, and—in the latter case—add β to the initially empty output set. Return the resulting output set. ◂

▸ **Proposition 17.** *Let q be an $\mathcal{E}$-query over output schema $\mathcal{F}$ and let $I \subseteq \mathcal{F}$. Query q is effectively safe relative to I iff each field in $\mathcal{F}$ is effectively safe in q relative to I.*

Proof. Let q be an $\mathcal{E}$-query over database schema $\mathcal{S}$ and output schema $\mathcal{F}$, and let $I \subseteq \mathcal{F}$.

Suppose that q is effectively safe relative to I. Then, there are uniformly computable approximations f_o of $q^{I \rightsquigarrow o}$ for all $o \in \mathcal{F}$. Let $f : \Lambda(\mathcal{E}) \times \mathrm{DB}(\mathcal{S}) \times \mathbb{V}^{I} \to 2^{\mathbb{V}^{\mathcal{F}}}$ be defined as

$$f(\lambda, D, \alpha) = \left\{\beta : \mathcal{F} \to \mathbb{V} \;\middle|\; \beta(o) \in f_o(\lambda, D, \alpha) \text{ for all } o \in \mathcal{F}\right\}.$$

It is routine to check that f is an overapproximation of $q^{I \rightsquigarrow \mathcal{F}}$ and that it is uniformly computable.

Conversely, suppose that there is a uniformly computable overapproximation f of $q^{I \rightsquigarrow \mathcal{F}}$. For each $o \in \mathcal{F}$, we can define a uniformly computable overapproximation f_o of $q^{I \rightsquigarrow o}$ by letting $f_o(\lambda, D, \alpha) = \left\{\beta(o) \;\middle|\; \beta \in f(\lambda, D, \alpha)\right\}$. ◂