Practical Relational Calculus Query Evaluation

Martin Raszyk
Department of Computer Science, ETH Zürich, Switzerland

David Basin
Department of Computer Science, ETH Zürich, Switzerland

Srđan Krstić
Department of Computer Science, ETH Zürich, Switzerland

Dmitriy Traytel
Department of Computer Science, University of Copenhagen, Denmark

Abstract

The relational calculus (RC) is a concise, declarative query language. However, existing RC query evaluation approaches are inefficient and often deviate from established algorithms based on finite tables used in database management systems. We devise a new translation of an arbitrary RC query into two safe-range queries, for which the finiteness of the query's evaluation result is guaranteed. Assuming an infinite domain, the two queries have the following meaning: The first is closed and characterizes the original query's relative safety, i.e., whether given a fixed database, the original query evaluates to a finite relation. The second safe-range query is equivalent to the original query, if the latter is relatively safe. We compose our translation with other, more standard ones to ultimately obtain two SQL queries. This allows us to use standard database management systems to evaluate arbitrary RC queries. We show that our translation improves the time complexity over existing approaches, which we also empirically confirm in both realistic and synthetic experiments.

2012 ACM Subject Classification Theory of computation → Database query languages (principles)

Keywords and phrases Relational calculus, relative safety, safe-range, query translation

Digital Object Identifier 10.4230/LIPIcs.ICDT.2022.11

Related Version *Full Version*: https://github.com/rc2sql/rc2sql

1 Introduction

Codd's theorem states that all domain-independent queries of the relational calculus (RC) can be expressed in relational algebra (RA) [10]. A popular interpretation of this result is that RA suffices to express all interesting queries. This interpretation justifies why SQL evolved as the practical database query language with the RA as its mathematical foundation. SQL is declarative and abstracts over the actual RA expression used to evaluate a query. Yet, SQL's syntax inherits RA's deliberate syntactic limitations, such as union-compatibility, which ensure domain independence. RC does not have such syntactic limitations, which arguably makes it a more attractive declarative query language than both RA and SQL. The main problem of RC is that it is not immediately clear how to evaluate even domain-independent queries, much less how to handle the domain-dependent (i.e., not domain-independent) ones.

As a running example, consider a shop in which brands (unary finite relation B of brands) sell products (binary finite relation P relating brands and products) and products are reviewed by users with a score (ternary finite relation S relating products, users, and scores). We consider a brand *suspicious* if there is a user and a score such that all the brand's products were reviewed by that user with that score. An RC query computing suspicious brands is

$$Q^{susp} := \mathsf{B}(b) \land \exists u, s.\ \forall p.\ \mathsf{P}(b, p) \longrightarrow \mathsf{S}(p, u, s).$$

25th International Conference on Database Theory (ICDT 2022).
Editors: Dan Olteanu and Nils Vortmeier; Article No. 11; pp. 11:1–11:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This query is domain-independent and follows closely our informal description. It is not, however, clear how to evaluate it because its second conjunct is domain-dependent as it is satisfied for every brand that does not occur in P. Finding suspicious brands using RA or SQL is a challenge, which only the best students from an undergraduate database course will accomplish. We give away an RA answer next (where $-$ is the set difference operator and $\triangleright$ is the anti-join, also known as the *generalized* difference operator [1]):

$$\pi_{brand}((\pi_{user,score}(\mathsf{S}) \times \mathsf{B}) - \pi_{brand,user,score}((\pi_{user,score}(\mathsf{S}) \times \mathsf{P}) \triangleright \mathsf{S})) \cup (\mathsf{B} - \pi_{brand}(\mathsf{P})).$$

The highlighted expressions $\pi_{user,score}(\mathsf{S})$ are called *generators*. They ensure that the left operands of the anti-join and set difference operators include or have the same columns (i.e., are union-compatible) as the corresponding right operands. (Following Codd [10], one could in principle also use the active domain to obtain canonical, but far less efficient, generators.)

Van Gelder and Topor [13, 14] present a translation from a decidable class of domain-independent RC queries, called *evaluable*, to RA expressions. Their translation of the evaluable Q^{susp} query would yield different generators, replacing both highlighted parts by $\pi_{user}(\mathsf{S}) \times \pi_{score}(\mathsf{S})$. That one can avoid this Cartesian product as shown above is subtle: Replacing only the first highlighted generator with the product results in an inequivalent RA expression.

Once we have identified suspicious brands, we may want to obtain the users whose scoring made the brands suspicious. In RC, omitting u's quantifier from Q^{susp} achieves just that:

$$Q^{susp}_{user} := \mathsf{B}(b) \wedge \exists s.\ \forall p.\ \mathsf{P}(b,p) \longrightarrow \mathsf{S}(p,u,s).$$

In contrast, RA cannot express the same property as it is domain-dependent (hence also not evaluable and thus out of scope for Van Gelder and Topor's translation): Q^{susp}_{user} is satisfied for every user if a brand has no products, i.e., it does not occur in P. Yet, Q^{susp}_{user} is satisfied for finitely many users on every database instance where P contains at least one row for every brand from the relation B, in other words Q^{susp}_{user} is *relatively safe* on such database instances.

How does one evaluate queries that are not evaluable or even domain-dependent? The main approaches from the literature (Section 2) are either to use variants of the active domain semantics [2, 5, 15] or to abandon finite relations entirely and evaluate queries using finite representations of infinite (but well-behaved) relations such as systems of constraints [26] or automatic structures [6]. These approaches favor expressiveness over efficiency. Unlike query translations, they cannot benefit from decades of practical database research and engineering.

In this work, we translate arbitrary RC queries to RA expressions under the assumption of an infinite domain. To deal with queries that are domain-dependent, our translation produces two RA expressions, instead of a single equivalent one. The first RA expression characterizes the original RC query's relative safety, the decidable question of whether the query evaluates to a finite relation for a given database, which can be the case even for a domain-dependent query, e.g., Q^{susp}_{user}. If the original query is relatively safe on a given database, i.e., produces some finite result, then the second RA expression evaluates to the same finite result. Taken together, the two RA expressions solve the *query capturability* problem [3]: they allow us to enumerate the original RC query's finite evaluation result, or to learn that it would be infinite using RA operations on the unmodified database.

Our translation of an RC query to two RA expressions proceeds in several steps via safe-range queries and the relational algebra normal form (Section 3). We focus on the first step of translating an RC query to two safe-range RC queries (Section 4), which fundamentally differs from Van Gelder and Topor's approach and produces better generators like $\pi_{user,score}(\mathsf{S})$. Our generators strictly improve the time complexity of query evaluation (Section 4.4).

After the more standard transformations to relational algebra normal form and from there to RA expressions, we translate the resulting RA expressions into SQL using the radb tool [30]. Along the way to SQL, we leverage various ideas from the literature to optimize the overall result (Section 6). For example, we generalize Claußen et al. [9]'s approach to avoid evaluating Cartesian products like $\pi_{user,score}(\mathsf{S}) \times \mathsf{P}$ in the above translation by using count aggregations.

The overall translation allows us to use standard database management systems to evaluate RC queries. We implement our translation and use PostgreSQL to evaluate the translated queries. Using a real Amazon review dataset [23] and our synthetic benchmark that generates hard database instances for random RC queries (Section 5), we evaluate our translation's performance. The evaluation shows that our approach outperforms Van Gelder and Topor's translation (which also uses PostgreSQL for evaluation) and other approaches (Section 6).

In summary, the following are our three main contributions:

- We devise a translation of an arbitrary RC query into a pair of RA expressions as described above. The time complexity of evaluating our translation's results improves upon Van Gelder and Topor's approach [14].
- We implement our translation and extend it to produce SQL queries. The resulting tool RC2SQL makes RC a viable input language for standard database management systems. We evaluate our tool on synthetic and real data and confirm that our translation's improved asymptotic time complexity carries over into practice.
- To challenge RC2SQL (and its competitors) in our evaluation, we devise the *Data Golf* benchmark that generates hard database instances for randomly generated RC queries.

2 Related Work

We recall Trakhtenbrot's theorem and the fundamental notions of *capturability* and *data complexity*. Given an RC query over a *finite* domain, Trakhtenbrot [27] showed that it is undecidable whether there exists a (finite) structure satisfying the query. In contrast, the question of whether a fixed structure satisfies the given RC query is decidable [2].

Kifer [16] calls a query class capturable if there is an algorithm that, given a query in the class and a database instance, enumerates the query's evaluation result, i.e., all tuples satisfying the query. Avron and Hirshfeld [3] observe that Kifer's notion is restricted because it requires every query in a capturable class to be domain independent. Hence, they propose an alternative definition that we also use: A query class is capturable if there is an algorithm that, given a query in the class, a (finite or infinite) domain, and a database instance, determines whether the query's evaluation result on the database instance over the domain is finite and enumerates the result in this case. Our work solves Avron and Hirshfeld's capturability problem additionally assuming an infinite domain.

Data complexity [29] is the complexity of recognizing if a tuple satisfies a fixed query over a database, as a function of the database size. Our capturability algorithm provides an upper bound on the data complexity for RC queries over an infinite domain that have a finite evaluation result (but it cannot decide if a tuple belongs to a query's result if the result is infinite).

Next, we group related approaches to evaluating RC queries into three categories.

Structure reduction. The classical approach to handling arbitrary RC queries is to evaluate them under a finite structure [18]. The core question here is whether the evaluation produces the same result as defined by the natural semantics, which typically considers infinite domains.

Codd's theorem [10] affirmatively answers this question for domain-independent queries, restricting the structure to the *active domain*. Ailamazyan et al. [2] show that RC is a capturable query class by extending the active domain with a few additional elements, whose number depends only on the query, and evaluating the query over this finite domain. *Natural–active collapse* results [5] generalize Ailamazyan et al.'s [2] result to extensions of RC (e.g., with order relations) by combining the structure reduction with a translation-based approach. Hull and Su [15] study several semantics of RC that guarantee the finiteness of the query's evaluation result. In particular, the "output-restricted unlimited interpretation" only restricts the query's evaluation result to tuples that only contain elements in the active domain, but the quantified variables still range over the (finite or infinite) underlying domain. Our work is inspired by all these theoretical landmarks, in particular Hull and Su's work (Section 4.1). Yet we avoid using (extended) active domains, which make query evaluation impractical.

Query translation. Another strategy is to translate a given query into one that can be evaluated efficiently, for example as a sequence of RA operations. Van Gelder and Topor pioneered this approach [13, 14] for RC. A core component of their translation is the choice of generators, which replace the active domain restrictions from structure reduction approaches and thereby improve the time complexity. Extensions to scalar and complex function symbols have also been studied [12, 19]. All these approaches focus on syntactic classes of RC, for which domain-independence is given, e.g., the *evaluable* queries of Van Gelder and Topor [14, Definition 5.2]. Our approach is inspired by Van Gelder and Topor's but generalizes it to handle arbitrary RC queries at the cost of assuming an infinite domain. Also, we further improve the time complexity of Van Gelder and Topor's approach by choosing better generators.

Evaluation with infinite relations. Constraint databases [26] obviate the need for using finite tables when evaluating RC queries. This yields significant expressiveness gains over RC. Yet the efficiency of the quantifier elimination procedures employed cannot compare with the simple evaluation of a projection operation in RA. Similarly, automatic structures [6] can represent the results of arbitrary RC queries finitely, but struggle with large quantities of data. We demonstrate this in our evaluation where we compare our translation to several modern incarnations of the above approaches, all based on binary decision diagrams [4, 7, 17, 20, 21].

3 Preliminaries

We introduce the RC syntax and semantics and define relevant classes of RC queries.

3.1 Relational Calculus

A signature σ is a triple $(\mathcal{C}, \mathcal{R}, \iota)$, where $\mathcal{C}$ and $\mathcal{R}$ are disjoint finite sets of constant and predicate symbols, and the function $\iota : \mathcal{R} \to \mathbb{N}$ maps each predicate symbol $r \in \mathcal{R}$ to its arity $\iota(r)$. Let $\sigma = (\mathcal{C}, \mathcal{R}, \iota)$ be a signature and $\mathcal{V}$ a countably infinite set of variables disjoint from $\mathcal{C} \cup \mathcal{R}$. The following grammar defines the syntax of RC queries:

$$Q ::= \bot \mid \top \mid x \approx t \mid r(t_1, \ldots, t_{\iota(r)}) \mid \neg Q \mid Q \vee Q \mid Q \wedge Q \mid \exists x.\, Q.$$

Here, $r \in \mathcal{R}$ is a predicate symbol, $t, t_1, \ldots, t_{\iota(r)} \in \mathcal{V} \cup \mathcal{C}$ are terms, and $x \in \mathcal{V}$ is a variable. We write $\exists \vec{v}.\, Q$ for $\exists v_1. \ldots \exists v_k.\, Q$ and $\forall \vec{v}.\, Q$ for $\neg \exists \vec{v}.\, \neg Q$, where $\vec{v}$ is a variable sequence $v_1, \ldots, v_k$. If $k = 0$, then both $\exists \vec{v}.\, Q$ and $\forall \vec{v}.\, Q$ denote just Q. Quantifiers have lower

$$
\begin{array}{lll|lll}
\multicolumn{3}{l|}{(\mathcal{S},\alpha) \not\models \bot;\ (\mathcal{S},\alpha) \models \top;} & (\mathcal{S},\alpha) \models (x \approx t) & \text{iff} & \alpha(x) = \alpha(t); \\
(\mathcal{S},\alpha) \models r(t_1,\ldots,t_{\iota(r)}) & \text{iff} & (\alpha(t_1),\ldots,\alpha(t_{\iota(r)})) \in r^{\mathcal{S}}; & (\mathcal{S},\alpha) \models (\neg Q) & \text{iff} & (\mathcal{S},\alpha) \not\models Q; \\
(\mathcal{S},\alpha) \models (Q_1 \vee Q_2) & \text{iff} & (\mathcal{S},\alpha) \models Q_1 \text{ or } (\mathcal{S},\alpha) \models Q_2; & (\mathcal{S},\alpha) \models (\exists x.\, Q) & \text{iff} & (\mathcal{S},\alpha[x \mapsto d]) \models Q, \\
(\mathcal{S},\alpha) \models (Q_1 \wedge Q_2) & \text{iff} & (\mathcal{S},\alpha) \models Q_1 \text{ and } (\mathcal{S},\alpha) \models Q_2; & & & \text{for some } d \in \mathcal{D}.
\end{array}
$$

Figure 1 The semantics of RC.

precedence than conjunctions and disjunctions, e.g., $\exists x.\, Q_1 \wedge Q_2$ means $\exists x.\, (Q_1 \wedge Q_2)$. We use $\approx$ to denote the equality of terms in RC to distinguish it from $=$, which denotes syntactic object identity. We also write $Q_1 \longrightarrow Q_2$ for $\neg Q_1 \vee Q_2$. However, defining $Q_1 \vee Q_2$ as a shorthand for $\neg(\neg Q_1 \wedge \neg Q_2)$ would complicate later definitions, e.g., the safe-range queries (Section 3.2).

We define the subquery partial order $\sqsubseteq$ on queries inductively on the structure of RC queries, e.g., Q_1 is a subquery of the query $Q_1 \wedge \neg \exists y.\, Q_2$. One can also view $\sqsubseteq$ as the (reflexive and transitive) subterm relation on the datatype of RC queries. We denote by $\mathsf{sub}(Q)$ the set of subqueries of a query Q and by $\mathsf{fv}(Q)$ the set of *free variables* in Q. Furthermore, we denote by $\vec{\mathsf{fv}}(Q)$ the sequence of free variables in Q based on some fixed ordering of variables. We lift this notation to sets of queries in the standard way. A query Q with no free variables, i.e., $\mathsf{fv}(Q) = \emptyset$, is called *closed*. Queries of the form $r(t_1,\ldots,t_{\iota(r)})$ and $x \approx \mathsf{c}$ are called *atomic predicates*. We define the predicate $\mathsf{ap}(\cdot)$ characterizing atomic predicates, i.e., $\mathsf{ap}(Q)$ is true iff Q is an atomic predicate. Queries of the form $\exists \vec{v}.\, r(t_1,\ldots,t_{\iota(r)})$ and $\exists \vec{v}.\, x \approx \mathsf{c}$ are called *quantified predicates*. We denote by $\tilde{\exists} x.\, Q$ the query obtained by existentially quantifying a variable x from a query Q if x is free in Q, i.e., $\tilde{\exists} x.\, Q := \exists x.\, Q$ if $x \in \mathsf{fv}(Q)$ and $\tilde{\exists} x.\, Q := Q$ otherwise. We lift this notation to sets of queries in the standard way. We use $\tilde{\exists} x.\, Q$ (instead of $\exists x.\, Q$) when constructing a query to avoid introducing bound variables that never occur in Q.

A structure $\mathcal{S}$ over a signature $(\mathcal{C},\mathcal{R},\iota)$ consists of a non-empty domain $\mathcal{D}$ and interpretations $\mathsf{c}^{\mathcal{S}} \in \mathcal{D}$ and $r^{\mathcal{S}} \subseteq \mathcal{D}^{\iota(r)}$, for each $\mathsf{c} \in \mathcal{C}$ and $r \in \mathcal{R}$. We assume that all the relations $r^{\mathcal{S}}$ are *finite*. Note that this assumption does *not* yield a finite structure (as defined in finite model theory [18]) since the domain $\mathcal{D}$ can still be infinite. A (*variable*) *assignment* is a mapping $\alpha : \mathcal{V} \to \mathcal{D}$. We additionally define α on constant symbols $\mathsf{c} \in \mathcal{C}$ as $\alpha(\mathsf{c}) = \mathsf{c}^{\mathcal{S}}$. We write $\alpha[x \mapsto d]$ for the assignment that maps x to $d \in \mathcal{D}$ and is otherwise identical to α. We lift this notation to sequences $\vec{x}$ and $\vec{d}$ of pairwise distinct variables and arbitrary domain elements of the same length. The semantics of RC queries for a structure $\mathcal{S}$ and an assignment α is defined in Figure 1. We write $\alpha \models Q$ for $(\mathcal{S},\alpha) \models Q$ if the structure $\mathcal{S}$ is fixed in the given context. For a fixed $\mathcal{S}$, only the assignments to Q's free variables influence $\alpha \models Q$, i.e., $\alpha \models Q$ is equivalent to $\alpha' \models Q$, for every variable assignment α' that agrees with α on $\mathsf{fv}(Q)$. For closed queries Q, we write $\models Q$ and say that Q holds, since closed queries either hold for all variable assignments or for none of them. We call a finite sequence $\vec{d}$ of domain elements $d_1, \ldots d_k \in \mathcal{D}$ a *tuple*. Given a query Q and a structure $\mathcal{S}$, we denote the set of satisfying tuples for Q by

$$[\![Q]\!]^{\mathcal{S}} = \{\vec{d} \in \mathcal{D}^{|\vec{\mathsf{fv}}(Q)|} \mid \text{there exists an assignment } \alpha \text{ such that } (\mathcal{S}, \alpha[\vec{\mathsf{fv}}(Q) \mapsto \vec{d}]) \models Q\}.$$

We omit $\mathcal{S}$ from $[\![Q]\!]^{\mathcal{S}}$ if $\mathcal{S}$ is fixed. We call the values from $[\![Q]\!]$ assigned to $x \in \mathsf{fv}(Q)$ column x.

The *active domain* $\mathsf{adom}^{\mathcal{S}}(Q)$ of a query Q and a structure $\mathcal{S}$ is a subset of the domain $\mathcal{D}$ containing the interpretations $\mathsf{c}^{\mathcal{S}}$ of all constant symbols that occur in Q and the values in the relations $r^{\mathcal{S}}$ interpreting all predicate symbols that occur in Q. Since $\mathcal{C}$ and $\mathcal{R}$ are finite and all $r^{\mathcal{S}}$ are finite relations of a finite arity $\iota(r)$, the active domain $\mathsf{adom}^{\mathcal{S}}(Q)$ is also a finite set. We omit $\mathcal{S}$ from $\mathsf{adom}^{\mathcal{S}}(Q)$ if $\mathcal{S}$ is fixed in the given context.

Queries Q_1 and Q_2 over the same signature are *equivalent*, written $Q_1 \equiv Q_2$, if $(\mathcal{S}, \alpha) \models Q_1 \iff (\mathcal{S}, \alpha) \models Q_2$, for every $\mathcal{S}$ and α. Queries Q_1 and Q_2 over the same signature are *inf-equivalent*, written $Q_1 \overset{\infty}{\equiv} Q_2$, if $(\mathcal{S}, \alpha) \models Q_1 \iff (\mathcal{S}, \alpha) \models Q_2$, for every $\mathcal{S}$ with an *infinite* domain $\mathcal{D}$ and every α. Clearly, equivalent queries are also inf-equivalent.

A query Q is *domain-independent* if $[\![Q]\!]^{\mathcal{S}_1} = [\![Q]\!]^{\mathcal{S}_2}$ holds for every two structures $\mathcal{S}_1$ and $\mathcal{S}_2$ that agree on the interpretations of constants ($\mathsf{c}^{\mathcal{S}_1} = \mathsf{c}^{\mathcal{S}_2}$) and predicates ($r^{\mathcal{S}_1} = r^{\mathcal{S}_2}$), while their domains $\mathcal{D}_1$ and $\mathcal{D}_2$ may differ. Agreement on the interpretations implies $\mathsf{adom}^{\mathcal{S}_1}(Q) = \mathsf{adom}^{\mathcal{S}_2}(Q) \subseteq \mathcal{D}_1 \cap \mathcal{D}_2$. It is undecidable whether an RC query is domain-independent [24, 28].

We denote by $Q[x \mapsto y]$ the query obtained from the query Q after replacing each free occurrence of the variable x by the variable y (possibly renaming bound variables to avoid capture) and performing constant propagation, i.e., simplifications like $(x \approx x) \equiv \top$, $Q \wedge \bot \equiv \bot$, $Q \vee \bot \equiv Q$, etc. We lift this notation to sets of queries in the standard way. Finally, we denote by $Q[x/\bot]$ the query obtained from Q after replacing every atomic predicate or equality containing a free variable x by $\bot$ (except for $x \approx x$) and performing constant propagation.

The function $\mathsf{flat}^{\oplus}(Q)$, where $\oplus \in \{\vee, \wedge\}$, computes a set of queries by "flattening" the operator $\oplus$: $\mathsf{flat}^{\oplus}(Q) \coloneqq \mathsf{flat}^{\oplus}(Q_1) \cup \mathsf{flat}^{\oplus}(Q_2)$ if $Q = Q_1 \oplus Q_2$ and $\mathsf{flat}^{\oplus}(Q) \coloneqq \{Q\}$ otherwise.

3.2 Safe-Range Queries

The class of *safe-range* queries [1] is a decidable subset of domain-independent RC queries. Its definition is based on the notion of range-restricted variables of a query. A variable is called *range-restricted* if "its possible values all lie within the active domain of the query" [1]. Intuitively, atomic predicates restrict the possible values of a variable that occurs in them as a term. An equality $x \approx y$ can extend the set of range-restricted variables in a conjunction $Q \wedge x \approx y$: If x or y is range-restricted in Q, then both x and y are range-restricted in $Q \wedge x \approx y$.

We formalize range-restricted variables using the *generated* relation $\mathsf{gen}(x, Q, \mathcal{G})$, defined in Figure 2. Specifically, $\mathsf{gen}(x, Q, \mathcal{G})$ holds if x is a range-restricted variable in Q and every satisfying assignment for Q satisfies some quantified predicate, referred to as *generator*, from $\mathcal{G}$. Note that, unlike in a similar definition by Van Gelder and Topor [14, Figure 5] that defines the rule $\mathsf{gen}(x, \exists y.\, Q_y, \mathcal{G})$ if $x \neq y$ and $\mathsf{gen}(x, Q_y, \mathcal{G})$, we modify the rule's conclusion to existentially quantify the bound variable y from all queries in $\mathcal{G}$ where y occurs: $\mathsf{gen}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G})$. Hence, $\mathsf{gen}(x, Q, \mathcal{G})$ implies $\mathsf{fv}(\mathcal{G}) \subseteq \mathsf{fv}(Q)$. We now formalize these relationships.

▶ **Lemma 1.** *Let Q be a query, $x \in \mathsf{fv}(Q)$, and $\mathcal{G}$ be a set of quantified predicates such that $\mathsf{gen}(x, Q, \mathcal{G})$. Then (i) for every $Q_{qp} \in \mathcal{G}$, we have $x \in \mathsf{fv}(Q_{qp})$ and $\mathsf{fv}(Q_{qp}) \subseteq \mathsf{fv}(Q)$, (ii) for every α such that $\alpha \models Q$, there exists $Q_{qp} \in \mathcal{G}$ such that $\alpha \models Q_{qp}$, and (iii) $Q[x/\bot] = \bot$.*

▶ **Definition 2.** *We define $\mathsf{gen}(x, Q)$ to hold iff there exists a set $\mathcal{G}$ such that $\mathsf{gen}(x, Q, \mathcal{G})$. Let $\mathsf{nongens}(Q) \coloneqq \{x \in \mathsf{fv}(Q) \mid \mathsf{gen}(x, Q)$ does not hold$\}$ be the set of free variables in a query Q that are not range-restricted. A query Q has* range-restricted free variables *if every free variable of Q is range-restricted, i.e., $\mathsf{nongens}(Q) = \emptyset$. A query Q has* range-restricted bound variables *if the bound variable y in every subquery $\exists y.\, Q_y$ of Q is range-restricted, i.e., $\mathsf{gen}(y, Q_y)$ holds. A query is* safe-range *if it has range-restricted free and range-restricted bound variables.*

$$\begin{array}{l}
\mathsf{gen}(x, \bot, \emptyset);\\
\mathsf{gen}(x, Q, \{Q\}) \text{ if } \mathsf{ap}(Q) \text{ and } x \in \mathsf{fv}(Q);\\
\mathsf{gen}(x, \neg\neg Q, \mathcal{G}) \text{ if } \mathsf{gen}(x, Q, \mathcal{G});\\
\mathsf{gen}(x, \neg(Q_1 \lor Q_2), \mathcal{G})\\
\quad \text{if } \mathsf{gen}(x, (\neg Q_1) \land (\neg Q_2), \mathcal{G});\\
\mathsf{gen}(x, \neg(Q_1 \land Q_2), \mathcal{G})\\
\quad \text{if } \mathsf{gen}(x, (\neg Q_1) \lor (\neg Q_2), \mathcal{G});\\
\mathsf{gen}(x, Q_1 \lor Q_2, \mathcal{G}_1 \cup \mathcal{G}_2)\\
\quad \text{if } \mathsf{gen}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{gen}(x, Q_2, \mathcal{G}_2);\\
\mathsf{gen}(x, Q_1 \land Q_2, \mathcal{G})\\
\quad \text{if } \mathsf{gen}(x, Q_1, \mathcal{G}) \text{ or } \mathsf{gen}(x, Q_2, \mathcal{G});\\
\mathsf{gen}(x, Q \land x \approx y, \mathcal{G}[y \mapsto x])\\
\quad \text{if } \mathsf{gen}(y, Q, \mathcal{G});\\
\mathsf{gen}(x, Q \land y \approx x, \mathcal{G}[y \mapsto x])\\
\quad \text{if } \mathsf{gen}(y, Q, \mathcal{G});\\
\mathsf{gen}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G})\\
\quad \text{if } x \neq y \text{ and } \mathsf{gen}(x, Q_y, \mathcal{G}).
\end{array}$$

Figure 2 The *generated* relation.

$$\begin{array}{ll}
\mathsf{cov}(x, x \approx x, \emptyset); & \\
\mathsf{cov}(x, Q, \emptyset) & \text{if } x \notin \mathsf{fv}(Q);\\
\mathsf{cov}(x, x \approx y, \{x \approx y\}) & \text{if } x \neq y;\\
\mathsf{cov}(x, y \approx x, \{x \approx y\}) & \text{if } x \neq y;\\
\mathsf{cov}(x, Q, \{Q\}) & \text{if } \mathsf{ap}(Q) \text{ and } x \in \mathsf{fv}(Q);\\
\mathsf{cov}(x, \neg Q, \mathcal{G}) & \text{if } \mathsf{cov}(x, Q, \mathcal{G});\\
\mathsf{cov}(x, Q_1 \lor Q_2, \mathcal{G}_1 \cup \mathcal{G}_2) & \text{if } \mathsf{cov}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{cov}(x, Q_2, \mathcal{G}_2);\\
\mathsf{cov}(x, Q_1 \lor Q_2, \mathcal{G}) & \text{if } \mathsf{cov}(x, Q_1, \mathcal{G}) \text{ and } Q_1[x/\bot] = \top;\\
\mathsf{cov}(x, Q_1 \lor Q_2, \mathcal{G}) & \text{if } \mathsf{cov}(x, Q_2, \mathcal{G}) \text{ and } Q_2[x/\bot] = \top;\\
\mathsf{cov}(x, Q_1 \land Q_2, \mathcal{G}_1 \cup \mathcal{G}_2) & \text{if } \mathsf{cov}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{cov}(x, Q_2, \mathcal{G}_2);\\
\mathsf{cov}(x, Q_1 \land Q_2, \mathcal{G}) & \text{if } \mathsf{cov}(x, Q_1, \mathcal{G}) \text{ and } Q_1[x/\bot] = \bot;\\
\mathsf{cov}(x, Q_1 \land Q_2, \mathcal{G}) & \text{if } \mathsf{cov}(x, Q_2, \mathcal{G}) \text{ and } Q_2[x/\bot] = \bot;\\
\multicolumn{2}{l}{\mathsf{cov}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G})}\\
\multicolumn{2}{l}{\qquad \text{if } x \neq y \text{ and } \mathsf{cov}(x, Q_y, \mathcal{G}) \text{ and } (x \approx y) \notin \mathcal{G};}\\
\multicolumn{2}{l}{\mathsf{cov}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G} \setminus \{x \approx y\} \cup \mathcal{G}_y[y \mapsto x])}\\
\multicolumn{2}{l}{\qquad \text{if } x \neq y \text{ and } \mathsf{cov}(x, Q_y, \mathcal{G}) \text{ and } \mathsf{gen}(y, Q_y, \mathcal{G}_y).}
\end{array}$$

Figure 3 The *covered* relation.

Relational algebra normal form (RANF) is a class of safe-range queries that can be easily mapped to RA and evaluated using the RA operations for projection, column duplication, selection, set union, binary join, and anti-join. Following a standard textbook [1, Section 5.4], we define the predicate $\mathsf{ranf}(\cdot)$ characterizing RANF queries and the translation $\mathsf{sr2ranf}(\cdot)$ of a safe-range query into an equivalent RANF query.

3.3 Query Cost

To assess the time complexity of evaluating a RANF query Q, we define the *cost* of Q over a structure $\mathcal{S}$, denoted $\mathsf{cost}^{\mathcal{S}}(Q)$, to be the sum of intermediate result sizes over all RANF subqueries of Q. Formally, $\mathsf{cost}^{\mathcal{S}}(Q) := \sum_{Q' \sqsubseteq Q,\ \mathsf{ranf}(Q')} \left| [\![Q']\!]^{\mathcal{S}} \right| \cdot |\mathsf{fv}(Q')|$. This corresponds to evaluating Q following its RANF structure using the RA operations. The complexity of these operations is linear in the combined input and output size (ignoring logarithmic factors due to set operations). The output size (the number of tuples times the number of variables) is counted in $\left| [\![Q']\!]^{\mathcal{S}} \right| \cdot |\mathsf{fv}(Q')|$ and the input size is counted as the output size for the input subqueries. Repeated subqueries are only considered once, which does not affect the asymptotics of query cost. In practice, the evaluation results for common subqueries can be reused.

4 Query Translation

Our approach to evaluating an arbitrary RC query Q over a fixed structure $\mathcal{S}$ with an infinite domain $\mathcal{D}$ proceeds by translating Q into a pair of safe-range queries $(Q_{\mathit{fin}}, Q_{\mathit{inf}})$ such that
(fv) $\mathsf{fv}(Q_{\mathit{fin}}) = \mathsf{fv}(Q)$ unless Q_{fin} is syntactically equal to $\bot$; $\mathsf{fv}(Q_{\mathit{inf}}) = \emptyset$;
(eval) $[\![Q]\!]$ is an infinite set if Q_{inf} holds; otherwise $[\![Q]\!] = [\![Q_{\mathit{fin}}]\!]$ is a finite set.
Since the queries Q_{fin} and Q_{inf} are safe-range, they are domain-independent and thus $[\![Q_{\mathit{fin}}]\!]$ is a finite set of tuples. In particular, $[\![Q]\!]$ is a finite set of tuples if Q_{inf} does not hold. Our translation generalizes Hull and Su's case distinction that restricts bound variables [15] to restrict all variables. Moreover, we use Van Gelder and Topor's idea to replace the active domain by a smaller set (generator) specific to each variable [14] while further improving the generators.

4.1 Restricting One Variable

Let x be a free variable in a query $\tilde{Q}$ with range-restricted bound variables. This assumption on $\tilde{Q}$ will be established by translating an arbitrary query Q bottom-up (Section 4.2). In this section, we develop a translation of $\tilde{Q}$ into an equivalent query $\tilde{Q}'$ that satisfies the following:

- $\tilde{Q}'$ has range-restricted bound variables;
- $\tilde{Q}'$ is a disjunction and x is range-restricted in all but the last disjunct.

The disjunct in which x is not range-restricted has a special form that is central to our translation: it is the conjunction of a query in which x does not occur and a query that is satisfied by infinitely many values of x. From the case distinction "for the corresponding variable: in or out of *adom*, and equality or inequality to other 'previous' variables if out of *adom*" [15], we translate $\tilde{Q}$ into the following equivalent query:

$$\begin{aligned}\tilde{Q} \equiv{} & (\tilde{Q} \land x \in \mathsf{adom}(\tilde{Q})) \lor \textstyle\bigvee_{y \in \mathsf{fv}(\tilde{Q}) \setminus \{x\}} (\tilde{Q}[x \mapsto y] \land x \approx y) \lor \\ & (\tilde{Q}[x/\bot] \land \neg(x \in \mathsf{adom}(\tilde{Q}) \lor \textstyle\bigvee_{y \in \mathsf{fv}(\tilde{Q}) \setminus \{x\}} x \approx y)).\end{aligned}$$

Here, $x \in \mathsf{adom}(\tilde{Q})$ stands for an RC query with a single free variable x that is satisfied by an assignment α if and only if $\alpha(x) \in \mathsf{adom}^{\mathcal{S}}(\tilde{Q})$. The translation distinguishes the following three cases for a fixed assignment α:

- if $\alpha(x) \in \mathsf{adom}^{\mathcal{S}}(\tilde{Q})$ holds, then we do not alter the query $\tilde{Q}$;
- if $x \approx y$ holds for some free variable $y \in \mathsf{fv}(\tilde{Q}) \setminus \{x\}$, then x can be replaced by y in $\tilde{Q}$;
- otherwise, $\tilde{Q}$ is equivalent to $\tilde{Q}[x/\bot]$, i.e., all atomic predicates with a free occurrence of x can be replaced by $\bot$ (because $\alpha(x) \notin \mathsf{adom}^{\mathcal{S}}(\tilde{Q})$), all equalities $x \approx y$ and $y \approx x$ for $y \in \mathsf{fv}(\tilde{Q}) \setminus \{x\}$ can be replaced by $\bot$ (because $\alpha(x) \neq \alpha(y)$), and all equalities $x \approx z$ for a bound variable z can be replaced by $\bot$ (because $\alpha(x) \notin \mathsf{adom}^{\mathcal{S}}(\tilde{Q})$ and z is range-restricted in its subquery $\exists z.\, Q_z$, by assumption, i.e., $\mathsf{gen}(z, Q_z)$ holds and thus, for all α', we have $\alpha' \models \exists z.\, Q_z$ if and only if there exists $d \in \mathsf{adom}^{\mathcal{S}}(Q_z) \subseteq \mathsf{adom}^{\mathcal{S}}(\tilde{Q})$ such that $\alpha'[z \mapsto d] \models Q_z$).

Note that $\exists \vec{\mathsf{fv}}(Q) \setminus \{x\}.\, Q$ is the query in which all free variables of Q except x are existentially quantified. Given a set of quantified predicates $\mathcal{G}$, we write $\exists \vec{\alpha}.\, \mathcal{G}$ for $\bigvee_{Q_{qp} \in \mathcal{G}} \exists \vec{\alpha}.\, Q_{qp}$. To avoid enumerating the entire active domain $\mathsf{adom}^{\mathcal{S}}(Q)$ of the query Q and a structure $\mathcal{S}$, Van Gelder and Topor [14] replace the condition $x \in \mathsf{adom}(Q)$ in their translation by $\exists \vec{\mathsf{fv}}(\mathcal{G}) \setminus \{x\}.\, \mathcal{G}$, where generator set $\mathcal{G}$ is a subset of atomic predicates. Because their translation [14] must yield an equivalent query (for every finite or infinite domain), $\mathcal{G}$ must satisfy, for all α,

$$\begin{array}{lll} \alpha \models \neg\exists \vec{\mathsf{fv}}(\mathcal{G}) \setminus \{x\}.\, \mathcal{G} & \Longrightarrow (\alpha \models Q \iff \alpha \models Q[x/\bot]) & (\textsc{vgt}_1) \quad \text{and} \\ \alpha \models Q[x/\bot] & \Longrightarrow \alpha \models \forall x.\, Q & (\textsc{vgt}_2). \end{array}$$

Note that ($\textsc{vgt}_2$) does not hold for the query $Q := \neg \mathsf{B}(x)$ and thus a generator set $\mathcal{G}$ of atomic predicates satisfying ($\textsc{vgt}_2$) only exists for a proper subset of all RC queries. In contrast, we only require that $\mathcal{G}$ satisfies ($\textsc{vgt}_1$) in our translation. To this end, we define a *covered* relation $\mathsf{cov}(x, Q, \mathcal{G})$ (in contrast to Van Gelder and Topor's *constrained* relation con [14, Figure 5]) such that, for every variable x and query $\tilde{Q}$ with range-restricted bound variables, there exists at least one set $\mathcal{G}$ such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ and ($\textsc{vgt}_1$) holds. Figure 3 shows the definition of this relation. Unlike the generator set $\mathcal{G}$ in $\mathsf{gen}(x, Q, \mathcal{G})$, the *cover* set $\mathcal{G}$ in $\mathsf{cov}(x, Q, \mathcal{G})$ may also contain equalities between two variables. Hence, we define a function $\mathsf{qps}(\mathcal{G})$ that collects all *generators*, i.e., quantified predicates and a function $\mathsf{eqs}(x, \mathcal{G})$ that collects all *variables* y distinct from x occurring in equalities of the form $x \approx y$. We use $\mathsf{qps}^{\vee}(\mathcal{G})$ to denote the query $\bigvee_{Q_{qp} \in \mathsf{qps}(\mathcal{G})} Q_{qp}$. We state the soundness and completeness of the relation $\mathsf{cov}(x, Q, \mathcal{G})$ in the next lemma, which follows by induction on the derivation of $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$.

▶ **Lemma 3.** *Let $\tilde{Q}$ be a query with range-restricted bound variables, $x \in \mathsf{fv}(\tilde{Q})$. Then there exists a set $\mathcal{G}$ of quantified predicates and equalities such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ holds and, for any such $\mathcal{G}$ and all α,*

$$\alpha \models \neg(\mathsf{qps}^{\vee}(\mathcal{G}) \vee \textstyle\bigvee_{y \in \mathsf{eqs}(x,\mathcal{G})} x \approx y) \implies (\alpha \models \tilde{Q} \iff \alpha \models \tilde{Q}[x/\bot]).$$

Finally, to preserve the dependencies between the variable x and the remaining free variables of Q occurring in the quantified predicates from $\mathsf{qps}(\mathcal{G})$, we do not project $\mathsf{qps}(\mathcal{G})$ on the single variable x, i.e., we restrict x by $\mathsf{qps}^{\vee}(\mathcal{G})$ instead of $\exists \vec{\mathsf{fv}}(Q) \setminus \{x\}.\, \mathsf{qps}(\mathcal{G})$. From Lemma 3, we derive our optimized translation characterized by the following lemma.

▶ **Lemma 4.** *Let $\tilde{Q}$ be a query with range-restricted bound variables, $x \in \mathsf{fv}(\tilde{Q})$, and $\mathcal{G}$ be such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ holds. Then $x \in \mathsf{fv}(Q_{qp})$ and $\mathsf{fv}(Q_{qp}) \subseteq \mathsf{fv}(\tilde{Q})$, for every $Q_{qp} \in \mathsf{qps}(\mathcal{G})$, and*

$$\begin{aligned}\tilde{Q} \equiv {} & (\tilde{Q} \wedge \mathsf{qps}^{\vee}(\mathcal{G})) \vee \textstyle\bigvee_{y \in \mathsf{eqs}(x,\mathcal{G})} (\tilde{Q}[x \mapsto y] \wedge x \approx y) \vee \\ & (\tilde{Q}[x/\bot] \wedge \neg(\mathsf{qps}^{\vee}(\mathcal{G}) \vee \textstyle\bigvee_{y \in \mathsf{eqs}(x,\mathcal{G})} x \approx y)).\end{aligned} \tag{★}$$

Note that x is not guaranteed to be range-restricted in (★)'s last disjunct. However, it occurs only in the negation of a disjunction of quantified predicates with a free occurrence of x and equalities of the form $x \approx \mathsf{c}$ or $x \approx y$. We will show how to handle such occurrences in Sections 4.2 and 4.3. Moreover, the negation of the disjunction can be omitted if (VGT$_2$) holds.

4.2 Restricting Bound Variables

Let x be a free variable in a query $\tilde{Q}$ with range-restricted bound variables. Suppose that the variable x is not range-restricted, i.e., $\mathsf{gen}(x, \tilde{Q})$ does not hold. To translate $\exists x.\, \tilde{Q}$ into an inf-equivalent query with range-restricted bound variables ($\exists x.\, \tilde{Q}$ does not have range-restricted bound variables precisely because x is not range-restricted in $\tilde{Q}$), we first apply (★) to $\tilde{Q}$ and distribute the existential quantifier binding x over disjunction. Next we observe that

$$\exists x.\, (\tilde{Q}[x \mapsto y] \wedge x \approx y) \equiv \tilde{Q}[x \mapsto y] \wedge \exists x.\, (x \approx y) \equiv \tilde{Q}[x \mapsto y],$$

where the first equivalence follows because x does not occur free in $\tilde{Q}[x \mapsto y]$ and the second equivalence follows from the straightforward validity of $\exists x.\, (x \approx y)$. Moreover, we observe that

$$\exists x.\, (\tilde{Q}[x/\bot] \wedge \neg(\mathsf{qps}^{\vee}(\mathcal{G}) \vee \textstyle\bigvee_{y \in \mathsf{eqs}(x,\mathcal{G})} x \approx y)) \overset{\infty}{\equiv} \tilde{Q}[x/\bot]$$

because x is not free in $\tilde{Q}[x/\bot]$ and there exists a value d for x in the infinite domain $\mathcal{D}$ such that $x \neq y$ holds for all finitely many $y \in \mathsf{eqs}(x, \mathcal{G})$ and d is not among the finitely many values interpreting the quantified predicates in $\mathsf{qps}(\mathcal{G})$. Altogether, we obtain the following lemma.

▶ **Lemma 5.** *Let $\tilde{Q}$ be a query with range-restricted bound variables, $x \in \mathsf{fv}(\tilde{Q})$, and $\mathcal{G}$ be a set of quantified predicates and equalities such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ holds. Then*

$$\exists x.\, \tilde{Q} \overset{\infty}{\equiv} (\exists x.\, \tilde{Q} \wedge \mathsf{qps}^{\vee}(\mathcal{G})) \vee \textstyle\bigvee_{y \in \mathsf{eqs}(x,\mathcal{G})} (\tilde{Q}[x \mapsto y]) \vee \tilde{Q}[x/\bot]. \tag{★∃}$$

Algorithm 1 Restricting bound variables.

```
    input:  An RC query $Q$.
    output: A query $\tilde{Q}$ with range-restricted bound variables such that $Q \overset{\infty}{\equiv} \tilde{Q}$.
1   function $\mathsf{fixbound}(\mathcal{Q}, x) =$
        $\{Q_{\mathit{fix}} \in \mathcal{Q} \mid x \in \mathsf{nongens}(Q_{\mathit{fix}})\}$;
2   function $\mathsf{rb}(Q) =$
3       switch $Q$ do
4           case $\neg Q'$ do return $\neg\mathsf{rb}(Q')$;
5           case $Q'_1 \vee Q'_2$ do return $\mathsf{rb}(Q'_1) \vee \mathsf{rb}(Q'_2)$;
6           case $Q'_1 \wedge Q'_2$ do return $\mathsf{rb}(Q'_1) \wedge \mathsf{rb}(Q'_2)$;
7           case $\exists x.\, Q_x$ do
8               $\mathcal{Q} := \mathsf{flat}^{\vee}(\mathsf{rb}(Q_x))$;
9               while $\mathsf{fixbound}(\mathcal{Q}, x) \neq \emptyset$ do
10                  $Q_{\mathit{fix}} \leftarrow \mathsf{fixbound}(\mathcal{Q}, x)$;
11                  $\mathcal{G} \leftarrow \{\mathcal{G} \mid \mathsf{cov}(x, Q_{\mathit{fix}}, \mathcal{G})\}$;
12                  $\mathcal{Q} := (\mathcal{Q} \setminus \{Q_{\mathit{fix}}\}) \cup$
                        $\{Q_{\mathit{fix}} \wedge \mathsf{qps}^{\vee}(\mathcal{G})\} \cup$
                        $\bigcup_{y \in \mathsf{eqs}(x,\mathcal{G})} \{Q_{\mathit{fix}}[x \mapsto y]\} \cup$
                        $\{Q_{\mathit{fix}}[x/\bot]\}$;
13              return $\bigvee_{\tilde{Q} \in \mathcal{Q}} \tilde{\exists} x.\, \tilde{Q}$;
14          otherwise do return $Q$;
```

Algorithm 2 Restricting free variables.

```
input:  An RC query $Q$.
output: Safe-range query pair $(Q_{\mathit{fin}}, Q_{\mathit{inf}})$ for which (fv) and (eval) hold.
function $\mathsf{fixfree}(\mathcal{Q}_{\mathit{fin}}) =$
    $\{(Q_{\mathit{fix}}, Q^{=}) \in \mathcal{Q}_{\mathit{fin}} \mid \mathsf{nongens}(Q_{\mathit{fix}}) \neq \emptyset\}$;
function $\mathsf{inf}(\mathcal{Q}_{\mathit{fin}}, Q) = \{(Q_{\not\infty}, Q^{=}) \in \mathcal{Q}_{\mathit{fin}} \mid$
    $\mathsf{disjointvars}(Q_{\not\infty}, Q^{=}) \neq \emptyset \vee$
    $\mathsf{fv}(Q_{\not\infty} \wedge Q^{=}) \neq \mathsf{fv}(Q)\}$;
function $\mathsf{split}(Q) =$
    $\mathcal{Q}_{\mathit{fin}} := \{(\mathsf{rb}(Q), \top)\}$; $\mathcal{Q}_{\mathit{inf}} := \emptyset$;
    while $\mathsf{fixfree}(\mathcal{Q}_{\mathit{fin}}) \neq \emptyset$ do
        $(Q_{\mathit{fix}}, Q^{=}) \leftarrow \mathsf{fixfree}(\mathcal{Q}_{\mathit{fin}})$;
        $x \leftarrow \mathsf{nongens}(Q_{\mathit{fix}})$;
        $\mathcal{G} \leftarrow \{\mathcal{G} \mid \mathsf{cov}(x, Q_{\mathit{fix}}, \mathcal{G})\}$;
        $\mathcal{Q}_{\mathit{fin}} := (\mathcal{Q}_{\mathit{fin}} \setminus \{(Q_{\mathit{fix}}, Q^{=})\}) \cup$
            $\{(Q_{\mathit{fix}} \wedge \mathsf{qps}^{\vee}(\mathcal{G}), Q^{=})\} \cup$
            $\bigcup_{y \in \mathsf{eqs}(x,\mathcal{G})} \{(Q_{\mathit{fix}}[x \mapsto y], Q^{=} \wedge x \approx y)\}$;
        $\mathcal{Q}_{\mathit{inf}} := \mathcal{Q}_{\mathit{inf}} \cup \{Q_{\mathit{fix}}[x/\bot]\}$;
    while $\mathsf{inf}(\mathcal{Q}_{\mathit{fin}}, Q) \neq \emptyset$ do
        $(Q_{\not\infty}, Q^{=}) \leftarrow \mathsf{inf}(\mathcal{Q}_{\mathit{fin}}, Q)$;
        $\mathcal{Q}_{\mathit{fin}} := \mathcal{Q}_{\mathit{fin}} \setminus \{(Q_{\not\infty}, Q^{=})\}$;
        $\mathcal{Q}_{\mathit{inf}} := \mathcal{Q}_{\mathit{inf}} \cup \{Q_{\not\infty} \wedge Q^{=}\}$;
    return $(\bigvee_{(Q_{\not\infty}, Q^{=}) \in \mathcal{Q}_{\mathit{fin}}} (Q_{\not\infty} \wedge Q^{=}),$
        $\mathsf{rb}(\bigvee_{Q_{\infty} \in \mathcal{Q}_{\mathit{inf}}} \exists \vec{\mathsf{fv}}(Q_{\infty}).\, Q_{\infty}))$;
```

Our approach for restricting all bound variables recursively applies Lemma 5. Because the set $\mathcal{G}$ such that $\mathsf{cov}(x, Q, \mathcal{G})$ holds is not necessarily unique, we introduce the following (general) notation. We denote the non-deterministic choice of an object X from a non-empty set $\mathcal{X}$ as $X \leftarrow \mathcal{X}$. We define the recursive function $\mathsf{rb}(Q)$ in Algorithm 1, where rb stands for *range-restrict bound* (variables). The function converts an arbitrary RC query Q into an inf-equivalent query with range-restricted bound variables. We proceed by describing the case $\exists x.\, Q_x$. First, $\mathsf{rb}(Q_x)$ is recursively applied on Line 8 to establish the precondition of Lemma 5 that the translated query has range-restricted bound variables. Because existential quantification distributes over disjunction, we flatten disjunction in $\mathsf{rb}(Q_x)$ and process the individual disjuncts independently. We apply ($\bigstar\exists$) to every disjunct Q_{fix} in which the variable x is not already range-restricted. For every Q'_{fix} added to $\mathcal{Q}$ after applying ($\bigstar\exists$) to Q_{fix} the variable x is either range-restricted or does not occur in Q'_{fix}, i.e., $x \notin \mathsf{nongens}(Q'_{\mathit{fix}})$. This entails the termination of the loop on Lines 9–12.

▶ **Example 6.** Consider the query $Q^{\mathit{susp}}_{\mathit{user}} := \mathsf{B}(b) \wedge \exists s.\, \forall p.\, \mathsf{P}(b, p) \longrightarrow \mathsf{S}(p, u, s)$ from Section 1. Restricting its bound variables yields the query

$$\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}}) = \mathsf{B}(b) \wedge ((\exists s.\, (\neg\exists p.\, \mathsf{P}(b, p) \wedge \neg\mathsf{S}(p, u, s)) \wedge (\exists p.\, \mathsf{S}(p, u, s))) \vee (\neg\exists p.\, \mathsf{P}(b, p))).$$

The bound variable p is already range-restricted in $Q^{\mathit{susp}}_{\mathit{user}}$ and thus only s must be restricted. Applying ($\bigstar$) to restrict s in $\neg\exists p.\, \mathsf{P}(b, p) \wedge \neg\mathsf{S}(p, u, s)$, then existentially quantifying s, and distributing the existential over disjunction yields the first disjunct in $\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}})$ above and $\exists s.\, (\neg\exists p.\, \mathsf{P}(b, p)) \wedge \neg(\exists p.\, \mathsf{S}(p, u, s))$ as the second disjunct. Because there exists some value in the infinite domain $\mathcal{D}$ that does not belong to the finite interpretation of the atomic predicate $\mathsf{S}(p, u, s)$, the query $\exists s.\, \neg(\exists p.\, \mathsf{S}(p, u, s))$ is a tautology over $\mathcal{D}$. Hence, $\exists s.\, (\neg\exists p.\, \mathsf{P}(b, p)) \wedge \neg(\exists p.\, \mathsf{S}(p, u, s))$ is inf-equivalent to $\neg\exists p.\, \mathsf{P}(b, p)$, i.e., the second disjunct in $\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}})$. This reasoning justifies applying ($\bigstar\exists$) to restrict s in $\exists s.\, \neg\exists p.\, \mathsf{P}(b, p) \wedge \neg\mathsf{S}(p, u, s)$.

4.3 Restricting Free Variables

Given an arbitrary query Q, we translate the inf-equivalent query $\mathsf{rb}(Q)$ with range-restricted bound variables into a pair of safe-range queries $(Q_{\mathit{fin}}, Q_{\mathit{inf}})$ such that our translation's main properties **(fv)** and **(eval)** hold. Our translation is based on the following lemma.

▶ **Lemma 7.** *Let a structure $\mathcal{S}$ with an infinite domain $\mathcal{D}$ be fixed. Let x be a free variable in a query $\tilde{Q}$ with range-restricted bound variables and let $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ for a set of quantified predicates and equalities $\mathcal{G}$. If $\tilde{Q}[x/\bot]$ is not satisfied by any tuple, then*

$$\llbracket\tilde{Q}\rrbracket = \left\llbracket(\tilde{Q} \wedge \mathsf{qps}^{\vee}(\mathcal{G})) \vee \bigvee\nolimits_{y \in \mathsf{eqs}(x,\mathcal{G})} (\tilde{Q}[x \mapsto y] \wedge x \approx y)\right\rrbracket. \qquad (☆)$$

If $\tilde{Q}[x/\bot]$ is satisfied by some tuple, then $\llbracket\tilde{Q}\rrbracket$ is an infinite set.

Proof. If $\tilde{Q}[x/\bot]$ is not satisfied by any tuple, then (☆) follows from (★). If $\tilde{Q}[x/\bot]$ is satisfied by some tuple, then the last disjunct in (★) applied to $\tilde{Q}$ is satisfied by infinitely many tuples obtained by assigning x some value from the infinite domain $\mathcal{D}$ such that $x \neq y$ holds for all finitely many $y \in \mathsf{eqs}(x, \mathcal{G})$ and x does not appear among the finitely many values interpreting the quantified predicates from $\mathsf{qps}(\mathcal{G})$. ◀

We remark that $\llbracket\tilde{Q}\rrbracket$ might be an infinite set of tuples even if $\tilde{Q}[x/\bot]$ is never satisfied, for some x. This is because $\tilde{Q}[y/\bot]$ might be satisfied by some tuple, for some y, in which case Lemma 7 (for y) implies that $\llbracket\tilde{Q}\rrbracket$ is an infinite set of tuples. Still, (☆) can be applied to $\tilde{Q}$ for x resulting in an equivalent query that is also satisfied by an infinite set of tuples.

Our approach is implemented by the function $\mathsf{split}(Q)$ defined in Algorithm 2. In the following, we describe this function and informally justify its correctness, formalized by the input/output specification. In $\mathsf{split}(Q)$, we represent the queries Q_{fin} and Q_{inf} using a set $\mathcal{Q}_{\mathit{fin}}$ of query pairs and a set $\mathcal{Q}_{\mathit{inf}}$ of queries such that

$$Q_{\mathit{fin}} \coloneqq \bigvee\nolimits_{(Q_{\not\infty}, Q^{=}) \in \mathcal{Q}_{\mathit{fin}}} (Q_{\not\infty} \wedge Q^{=}), \qquad Q_{\mathit{inf}} \coloneqq \bigvee\nolimits_{Q_{\infty} \in \mathcal{Q}_{\mathit{inf}}} \exists \vec{\mathsf{fv}}(Q_{\infty}).\, Q_{\infty},$$

and, for every $(Q_{\not\infty}, Q^{=}) \in \mathcal{Q}_{\mathit{fin}}$, $Q^{=}$ is a conjunction of equalities. As long as there exists some $(Q_{\mathit{fix}}, Q^{=}) \in \mathcal{Q}_{\mathit{fin}}$ such that $\mathsf{nongens}(Q_{\mathit{fix}}) \neq \emptyset$, we apply (☆) to Q_{fix} and add the query $Q_{\mathit{fix}}[x/\bot]$ to $\mathcal{Q}_{\mathit{inf}}$. We remark that if we applied (☆) to the entire disjunct $Q_{\mathit{fix}} \wedge Q^{=}$, the loop on Lines 5–10 might not terminate. Note that, for every $(Q'_{\mathit{fix}}, Q'^{=})$ added to $\mathcal{Q}_{\mathit{fin}}$ after applying (☆) to Q_{fix}, $\mathsf{nongens}(Q'_{\mathit{fix}})$ is a proper subset of $\mathsf{nongens}(Q_{\mathit{fix}})$. This entails the termination of the loop on Lines 5–10. Finally, if $\llbracket Q_{\mathit{fix}}\rrbracket$ is an infinite set of tuples, then $\llbracket Q_{\mathit{fix}} \wedge Q^{=}\rrbracket$ is an infinite set of tuples, too. This is because the equalities in $Q^{=}$ merely duplicate columns of the query Q_{fix}. Hence, it indeed suffices to apply (☆) to Q_{fix} instead of $Q_{\mathit{fix}} \wedge Q^{=}$.

After the loop on Lines 5–10 in Algorithm 2 terminates, for every $(Q_{\not\infty}, Q^{=}) \in \mathcal{Q}_{\mathit{fin}}$, $Q_{\not\infty}$ is a safe-range query and $Q^{=}$ is a conjunction of equalities such that $\mathsf{fv}(Q_{\not\infty} \wedge Q^{=}) = \mathsf{fv}(Q)$. However, the query $Q_{\not\infty} \wedge Q^{=}$ need not be safe-range, e.g., if $Q_{\not\infty} \coloneqq \mathsf{B}(x)$ and $Q^{=} \coloneqq (x \approx y \wedge u \approx v)$. Given a set of equalities $\mathcal{Q}^{=}$, let $\mathsf{classes}(\mathcal{Q}^{=})$ be the set of equivalence classes of free variables $\mathsf{fv}(\mathcal{Q}^{=})$ with respect to $\mathcal{Q}^{=}$. For instance, $\mathsf{classes}(\{x \approx y, y \approx z, u \approx v\}) = \{\{x, y, z\}, \{u, v\}\}$. Let $\mathsf{disjointvars}(Q_{\not\infty}, Q^{=}) \coloneqq \bigcup_{V \in \mathsf{classes}(\mathsf{flat}^{\wedge}(Q^{=})), V \cap \mathsf{fv}(Q_{\not\infty}) = \emptyset} V$ be the set of all variables in equivalence classes from $\mathsf{classes}(\mathsf{flat}^{\wedge}(Q^{=}))$ that are disjoint from $Q_{\not\infty}$'s free variables. Then, $Q_{\not\infty} \wedge Q^{=}$ is safe-range if and only if $\mathsf{disjointvars}(Q_{\not\infty}, Q^{=}) = \emptyset$ (recall the definition of safe-range).

Now if $\mathsf{disjointvars}(Q_{\not\infty}, Q^{=}) \neq \emptyset$ and $Q_{\not\infty} \wedge Q^{=}$ is satisfied by some tuple, then $\llbracket Q_{\not\infty} \wedge Q^{=}\rrbracket$ is an infinite set of tuples because all equivalence classes of variables in $\mathsf{disjointvars}(Q_{\not\infty}, Q^{=}) \neq \emptyset$ can be assigned arbitrary values from the infinite domain $\mathcal{D}$. In our example with

$Q_{\not\infty} := \mathsf{B}(x)$ and $Q^{=} := (x \approx y \wedge u \approx v)$, we have $\mathsf{disjointvars}(Q_{\not\infty}, Q^{=}) = \{u, v\} \neq \emptyset$. Moreover, if $\mathsf{fv}(Q_{\not\infty} \wedge Q^{=}) \neq \mathsf{fv}(Q)$ and $Q_{\not\infty} \wedge Q^{=}$ is satisfied by some tuple, then this tuple can be extended to infinitely many tuples over $\mathsf{fv}(Q)$ by choosing arbitrary values from the infinite domain $\mathcal{D}$ for the variables in the non-empty set $\mathsf{fv}(Q) \setminus \mathsf{fv}(Q_{\not\infty} \wedge Q^{=})$. Hence, for every $(Q_{\not\infty}, Q^{=}) \in \mathcal{Q}_{fin}$ with $\mathsf{disjointvars}(Q_{\not\infty}, Q^{=}) \neq \emptyset$ or $\mathsf{fv}(Q_{\not\infty} \wedge Q^{=}) \neq \mathsf{fv}(Q)$, we remove $(Q_{\not\infty}, Q^{=})$ from $\mathcal{Q}_{fin}$ and add $Q_{\not\infty} \wedge Q^{=}$ to $\mathcal{Q}_{inf}$. Note that we only remove pairs from $\mathcal{Q}_{fin}$, hence, the loop on Lines 11–14 terminates. Afterwards, the query Q_{fin} is safe-range. However, the query Q_{inf} need not be safe-range. Indeed, every query $Q_{\infty} \in \mathcal{Q}_{inf}$ has range-restricted bound variables, but not all the free variables of Q_{∞} need be range-restricted and thus the query $\exists \vec{\mathsf{fv}}(Q_{\infty}).\, Q_{\infty}$ need not be safe-range. But the query Q_{inf} is closed and thus the inf-equivalent query $\mathsf{rb}(Q_{inf})$ with range-restricted bound variables is safe-range.

▶ **Lemma 8.** *Let Q be an RC query and $\mathsf{split}(Q) = (Q_{fin}, Q_{inf})$. Then the queries Q_{fin} and Q_{inf} are safe-range; $\mathsf{fv}(Q_{fin}) = \mathsf{fv}(Q)$ unless Q_{fin} is syntactically equal to $\bot$; and $\mathsf{fv}(Q_{inf}) = \emptyset$.*

▶ **Lemma 9.** *Let a structure $\mathcal{S}$ with an infinite domain $\mathcal{D}$ be fixed. Let Q be an RC query and $\mathsf{split}(Q) = (Q_{fin}, Q_{inf})$. If $\models Q_{inf}$, then $[\![Q]\!]$ is an infinite set. Otherwise, $[\![Q]\!] = [\![Q_{fin}]\!]$ is a finite set.*

By Lemma 8, Q_{fin} is a safe-range (and thus also domain-independent) query. Hence, for a fixed structure $\mathcal{S}$, the tuples in $[\![Q_{fin}]\!]$ only contain elements in the active domain $\mathsf{adom}(Q_{fin})$, i.e., $[\![Q_{fin}]\!] = [\![Q_{fin}]\!] \cap \mathsf{adom}(Q_{fin})^{|\mathsf{fv}(Q_{fin})|}$. Our translation does not introduce new constants in Q_{fin} and thus $\mathsf{adom}(Q_{fin}) \subseteq \mathsf{adom}(Q)$. Hence, by Lemma 9, if $\not\models Q_{inf}$, then $[\![Q_{fin}]\!]$ is equal to the "output-restricted unlimited interpretation" [15] of Q, i.e., $[\![Q_{fin}]\!] = [\![Q]\!] \cap \mathsf{adom}(Q)^{|\mathsf{fv}(Q)|}$. In contrast, if $\models Q_{inf}$, then $[\![Q_{fin}]\!] = [\![Q]\!] \cap \mathsf{adom}(Q)^{|\mathsf{fv}(Q)|}$ does not necessarily hold. For instance, for $Q := \neg \mathsf{B}(x)$, our translation yields $\mathsf{split}(Q) = (\bot, \top)$. In this case, we have $Q_{inf} = \top$ and thus $\models Q_{inf}$ because $\neg \mathsf{B}(x)$ is satisfied by infinitely many tuples over an infinite domain. However, if $\mathsf{B}(x)$ is never satisfied, then $[\![Q_{fin}]\!] = \emptyset$ is not equal to $[\![Q]\!] \cap \mathsf{adom}(Q)^{|\mathsf{fv}(Q)|}$.

▶ **Example 10.** Consider the query $Q := \mathsf{B}(x) \vee \mathsf{P}(x, y)$. The variable y is not range-restricted in Q and thus $\mathsf{split}(Q)$ restricts y by a conjunction of Q with $\mathsf{P}(x, y)$. However, if $Q[y/\bot] = \mathsf{B}(x)$ is satisfied by some tuple, then $[\![Q]\!]$ contains infinitely many tuples. Hence, $\mathsf{split}(Q) = ((\mathsf{B}(x) \vee \mathsf{P}(x, y)) \wedge \mathsf{P}(x, y), \exists x.\, \mathsf{B}(x))$. Because $Q_{fin} = (\mathsf{B}(x) \vee \mathsf{P}(x, y)) \wedge \mathsf{P}(x, y)$ is only used if $\not\models Q_{inf}$, i.e., if $\mathsf{B}(x)$ is never satisfied, we could simplify Q_{fin} to $\mathsf{P}(x, y)$. However, our translation does not implement such heuristic simplifications.

▶ **Example 11.** Consider the query $Q := \mathsf{B}(x) \wedge u \approx v$. The variables u and v are not range-restricted in Q and thus $\mathsf{split}(Q)$ chooses one of these variables (e.g., u) and restricts it by splitting Q into $Q_{\not\infty} = \mathsf{B}(x)$ and $Q^{=} = u \approx v$. Now, all variables are range-restricted in $Q_{\not\infty}$, but the variables in $Q_{\not\infty}$ and $Q^{=}$ are disjoint. Hence, $[\![Q]\!]$ contains infinitely many tuples whenever $Q_{\not\infty}$ is satisfied by some tuple. In contrast, $[\![Q]\!] = \emptyset$ if $Q_{\not\infty}$ is never satisfied. Hence, we have $\mathsf{split}(Q) = (\bot, \exists x.\, \mathsf{B}(x))$.

▶ **Example 12.** Consider the query $Q^{susp}_{user} := \mathsf{B}(b) \wedge \exists s.\, \forall p.\, \mathsf{P}(b, p) \longrightarrow \mathsf{S}(p, u, s)$ from Section 1. Restricting its bound variables yields the query $\mathsf{rb}(Q^{susp}_{user}) = \mathsf{B}(b) \wedge ((\exists s.\, (\neg \exists p.\, \mathsf{P}(b, p) \wedge \neg \mathsf{S}(p, u, s)) \wedge (\exists p.\, \mathsf{S}(p, u, s))) \vee (\neg \exists p.\, \mathsf{P}(b, p)))$ derived in Example 6. Splitting Q^{susp}_{user} yields

$$\mathsf{split}(Q^{susp}_{user}) = (\mathsf{rb}(Q^{susp}_{user}) \wedge (\exists s, p.\, \mathsf{S}(p, u, s)), \exists b.\, \mathsf{B}(b) \wedge \neg \exists p.\, \mathsf{P}(b, p)).$$

To understand $\mathsf{split}(Q^{susp}_{user})$, we apply $(\bigstar)$ to $\mathsf{rb}(Q^{susp}_{user})$ for the free variable u:

$$\mathsf{rb}(Q^{susp}_{user}) \equiv (\mathsf{rb}(Q^{susp}_{user}) \wedge (\exists s, p.\, \mathsf{S}(p, u, s))) \vee (\mathsf{B}(b) \wedge (\neg \exists p.\, \mathsf{P}(b, p)) \wedge \neg \exists s, p.\, \mathsf{S}(p, u, s)).$$

If the subquery $\mathsf{B}(b) \wedge (\neg\exists p.\, \mathsf{P}(b, p))$ from the second disjunct is satisfied for some b, then Q^{susp}_{user} is satisfied by infinitely many values for u from the infinite domain $\mathcal{D}$ that do not belong to the finite interpretation of $\mathsf{S}(p, u, s)$ and thus satisfy the subquery $\neg\exists s, p.\, \mathsf{S}(p, u, s)$. Hence, $[\![Q^{susp}_{user}]\!]^{\mathcal{S}} = [\![\mathsf{rb}(Q^{susp}_{user})]\!]^{\mathcal{S}}$ is an infinite set of tuples whenever $\mathsf{B}(b) \wedge \neg\exists p.\, \mathsf{P}(b, p)$ is satisfied for some b. In contrast, if $\mathsf{B}(b) \wedge \neg\exists p.\, \mathsf{P}(b, p)$ is not satisfied for any b, then Q^{susp}_{user} is equivalent to $\mathsf{rb}(Q^{susp}_{user}) \wedge (\exists s, p.\, \mathsf{S}(p, u, s))$ obtained also by applying (☆) to Q^{susp}_{user} for the free variable u.

▶ **Definition 13.** *Let Q be an RC query and $\mathsf{split}(Q) = (Q_{fin}, Q_{inf})$. Let $\hat{Q}_{fin} := \mathsf{sr2ranf}(Q_{fin})$ and $\hat{Q}_{inf} := \mathsf{sr2ranf}(Q_{inf})$ be the equivalent RANF queries. We define $\mathsf{rw}(Q) := (\hat{Q}_{fin}, \hat{Q}_{inf})$.*

4.4 Complexity Analysis

In this section, we analyze the time complexity of capturing Q, i.e., checking if $[\![Q]\!]$ is finite and enumerating $[\![Q]\!]$ if it is finite. To bound the asymptotic time complexity of capturing a fixed Q, we ignore the (constant) time complexity of computing $\mathsf{rw}(Q) = (\hat{Q}_{fin}, \hat{Q}_{inf})$ and focus on the time complexity of evaluating the RANF queries $\hat{Q}_{fin}$ and $\hat{Q}_{inf}$, i.e., the query cost of $\hat{Q}_{fin}$ and $\hat{Q}_{inf}$. Without loss of generality, we assume that the input query Q has pairwise distinct (free and bound) variables to derive a set of quantified predicates from Q's atomic predicates and formulate our time complexity bound. Nevertheless, the RANF queries $\hat{Q}_{fin}$ and $\hat{Q}_{inf}$ computed by our translation need not have pairwise distinct (free and bound) variables.

Let $\mathsf{av}(Q)$ be the set of all (free and bound) variables in a query Q. We define the relation $\precsim_Q$ on $\mathsf{av}(Q)$ such that $x \precsim_Q y$ iff the scope of an occurrence of $x \in \mathsf{av}(Q)$ is contained in the scope of an occurrence of $y \in \mathsf{av}(Q)$. Formally, we define $x \precsim_Q y$ iff $y \in \mathsf{fv}(Q)$ or $\exists x.\, Q_x \sqsubseteq \exists y.\, Q_y \sqsubseteq Q$ for some Q_x and Q_y. Note that $\precsim_Q$ is a preorder on all variables and a partial order on the bound variables for every query with pairwise distinct (free and bound) variables.

Let $\mathsf{aps}(Q)$ be the set of all atomic predicates in a query Q. We denote by $\overline{\mathsf{qps}}(Q)$ the set of quantified predicates obtained from $\mathsf{aps}(Q)$ by performing the variable substitution $x \mapsto y$, where x and y are related by equalities in Q and $x \precsim_Q y$, and existentially quantifying from a quantified predicate Q_{qp} the innermost bound variable x in Q that is free in Q_{qp}. Let $\mathsf{eqs}^*(Q)$ be the transitive closure of equalities occurring in Q. Formally, we define $\overline{\mathsf{qps}}(Q)$ by:

- $Q_{ap} \in \overline{\mathsf{qps}}(Q)$ if $Q_{ap} \in \mathsf{aps}(Q)$;
- $Q_{qp}[x \mapsto y] \in \overline{\mathsf{qps}}(Q)$ if $Q_{qp} \in \overline{\mathsf{qps}}(Q)$, $(x, y) \in \mathsf{eqs}^*(Q)$, and $x \precsim_Q y$;
- $\exists x.\, Q_{qp} \in \overline{\mathsf{qps}}(Q)$ if $Q_{qp} \in \overline{\mathsf{qps}}(Q)$, $x \in \mathsf{fv}(Q_{qp}) \setminus \mathsf{fv}(Q)$, and $x \precsim_Q y$ for all $y \in \mathsf{fv}(Q_{qp})$.

We bound the complexity of capturing Q by considering subsets $\mathcal{Q}_{qps}$ of quantified predicates $\overline{\mathsf{qps}}(Q)$ that are *minimal* in the sense that every quantified predicate in $\mathcal{Q}_{qps}$ contains a unique free variable that is not free in any other quantified predicate in $\mathcal{Q}_{qps}$. Formally, we define $\mathsf{minimal}(\mathcal{Q}_{qps}) := \forall Q_{qp} \in \mathcal{Q}_{qps}.\, \mathsf{fv}(\mathcal{Q}_{qps} \setminus \{Q_{qp}\}) \neq \mathsf{fv}(\mathcal{Q}_{qps})$. Every minimal subset $\mathcal{Q}_{qps}$ of quantified predicates $\overline{\mathsf{qps}}(Q)$ contributes the product of the numbers of tuples satisfying each quantified predicate $Q_{qp} \in \mathcal{Q}_{qps}$ to the overall bound (that product is an upper bound on the number of tuples satisfying the join over all $Q_{qp} \in \mathcal{Q}_{qps}$). Similarly to Ngo et al. [22], we use the notation $\tilde{\mathcal{O}}(\cdot)$ to hide logarithmic factors incurred by set operations.

▶ **Theorem 14.** *Let Q be a fixed RC query with pairwise distinct (free and bound) variables. The time complexity of capturing Q, i.e., checking if $[\![Q]\!]$ is finite and enumerating $[\![Q]\!]$ if it is finite, is in $\tilde{\mathcal{O}}\left(\sum_{\mathcal{Q}_{qps} \subseteq \overline{\mathsf{qps}}(Q), \mathsf{minimal}(\mathcal{Q}_{qps})} \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]|\right)$.*

We prove Theorem 14 in our extended report [25]. Examples 15 and 16 show that the time complexity from Theorem 14 cannot be achieved by the translation of Van Gelder and Topor [14] or over finite domains. Example 17 shows how equalities affect the bound in Theorem 14.

▶ **Example 15.** Consider the query $Q \coloneqq \mathsf{B}(b) \wedge \exists u, s.\, \neg\exists p.\, \mathsf{P}(b,p) \wedge \neg\mathsf{S}(p,u,s)$, equivalent to Q^{susp} from Section 1. Then $\mathsf{aps}(Q) = \{\mathsf{B}(b), \mathsf{P}(b,p), \mathsf{S}(p,u,s)\}$ and $\overline{\mathsf{qps}}(Q) = \{\mathsf{B}(b), \mathsf{P}(b,p), \exists p.\, \mathsf{P}(b,p), \mathsf{S}(p,u,s), \exists p.\, \mathsf{S}(p,u,s), \exists s,p.\, \mathsf{S}(p,u,s), \exists u,s,p.\, \mathsf{S}(p,u,s)\}$. The translated query Q_{vgt} by Van Gelder and Topor [14] restricts the variables r and s by $\exists s,p.\, \mathsf{S}(p,u,s)$ and $\exists u,p.\, \mathsf{S}(p,u,s)$, respectively. For an interpretation of B by $\{(\mathsf{c}') \mid \mathsf{c}' \in \{1,\dots,n\}\}$, P by $\{(\mathsf{c}',\mathsf{c}') \mid \mathsf{c}' \in \{1,\dots,n\}\}$, and S by $\{(\mathsf{c},\mathsf{c}',\mathsf{c}') \mid \mathsf{c} \in \{1,\dots,n\}, \mathsf{c}' \in \{1,\dots,m\}\}$, $n,m \in \mathbb{N}$, computing the join of $\mathsf{P}(b,p)$, $\exists s,p.\, \mathsf{S}(p,u,s)$, and $\exists u,p.\, \mathsf{S}(p,u,s)$, which is a Cartesian product, results in a time complexity in $\Omega(n \cdot m^2)$ for Q_{vgt}. In contrast, Theorem 14 yields an asymptotically better time complexity in $\tilde{\mathcal{O}}\left(n+m+n\cdot m\right)$ for our translation:

$$\tilde{\mathcal{O}}\left(|[\![\mathsf{B}(b)]\!]| + |[\![\mathsf{P}(b,p)]\!]| + |[\![\mathsf{S}(p,u,s)]\!]| + (|[\![\mathsf{B}(b)]\!]| + |[\![\mathsf{P}(b,p)]\!]|) \cdot |[\![\mathsf{S}(p,u,s)]\!]|\right).$$

▶ **Example 16.** The query $\neg\mathsf{S}(x,y,z)$ is satisfied by a finite set of tuples over a finite domain $\mathcal{D}$ (as is every other query over a finite domain). For an interpretation of S by $\{(\mathsf{c},\mathsf{c},\mathsf{c}) \mid \mathsf{c} \in \mathcal{D}\}$, the equality $|\mathcal{D}| = |[\![\mathsf{S}(x,y,z)]\!]|$ holds and the number of satisfying tuples is

$$|[\![\neg\mathsf{S}(x,y,z)]\!]| = |\mathcal{D}|^3 - |[\![\mathsf{S}(x,y,z)]\!]| = |[\![\mathsf{S}(x,y,z)]\!]|^3 - |[\![\mathsf{S}(x,y,z)]\!]| \in \Omega(|[\![\mathsf{S}(x,y,z)]\!]|^3),$$

which exceeds the bound $\tilde{\mathcal{O}}\left(|[\![\mathsf{S}(x,y,z)]\!]|\right)$ of Theorem 14. Hence, our infinite domain assumption is crucial for achieving the better complexity bound.

▶ **Example 17.** Consider the following query over the domain $\mathcal{D} = \mathbb{N}$ of natural numbers:

$$\begin{aligned} Q \coloneqq \forall u.\, (u \approx 0 \vee u \approx 1 \vee u \approx 2) \longrightarrow &\\ (\exists v.\, \mathsf{B}(v) \wedge (u \approx 0 \longrightarrow x \approx v) \wedge (u \approx 1 \longrightarrow y \approx v) \wedge (u \approx 2 \longrightarrow z \approx v)).& \end{aligned}$$

Note that this query is equivalent to $Q \equiv \mathsf{B}(x) \wedge \mathsf{B}(y) \wedge \mathsf{B}(z)$ and thus it is satisfied by a finite set of tuples of size $|[\![\mathsf{B}(x)]\!]| \cdot |[\![\mathsf{B}(y)]\!]| \cdot |[\![\mathsf{B}(z)]\!]| = |[\![\mathsf{B}(x)]\!]|^3$. The set of atomic predicates of Q is $\mathsf{aps}(Q) = \{\mathsf{B}(v)\}$ and it must be closed under the equalities occurring in Q to yield a valid bound in Theorem 14. In this case, $\overline{\mathsf{qps}}(Q) = \{\mathsf{B}(v), \exists v.\, \mathsf{B}(v), \mathsf{B}(x), \mathsf{B}(y), \mathsf{B}(z)\}$ and the bound in Theorem 14 is $|[\![\mathsf{B}(v)]\!]| \cdot |[\![\mathsf{B}(x)]\!]| \cdot |[\![\mathsf{B}(y)]\!]| \cdot |[\![\mathsf{B}(z)]\!]| = |[\![\mathsf{B}(x)]\!]|^4$. In particular, this bound is not tight, but it still reflects the complexity of evaluating the RANF queries produced by our translation as it does not derive the equivalence $Q \equiv \mathsf{B}(x) \wedge \mathsf{B}(y) \wedge \mathsf{B}(z)$.

5 Data Golf Benchmark

In this section, we devise the *Data Golf* benchmark for generating structures for given RC queries. We will use the benchmark in our empirical evaluation (Section 6). Given an RC query, we seek a structure that results in a nontrivial evaluation result for the overall query and for all its subqueries. Intuitively, the resulting structure makes query evaluation potentially more challenging compared to the case where some subquery results in a trivial (e.g., empty) evaluation result. More specifically, Data Golf has two objectives. The first resembles the *regex golf* game's objective [11] (hence the name) and aims to find a structure on which the result of a given query contains a given *positive* set of tuples and does not contain any tuples from another given *negative* set. The second objective is to ensure that all the query's subqueries evaluate to a non-trivial result.

Algorithm 3 Computing the Data Golf structure.

```
input: An RC query $Q$ with pairwise distinct (free and bound) variables satisfying
       CON, CST, VAR, REP, a sequence of distinct variables $\vec{v}$, $\mathsf{fv}(Q) \subseteq \vec{v}$, sets of
       tuples $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$ over $\vec{v}$ such that $\left|\mathcal{T}^+_{\vec{v}}[x]\right| = \left|\mathcal{T}^+_{\vec{v}}\right|$, $\left|\mathcal{T}^-_{\vec{v}}[x]\right| = \left|\mathcal{T}^-_{\vec{v}}\right|$, and
       $\mathcal{T}^+_{\vec{v}}[x] \cap \mathcal{T}^-_{\vec{v}}[x] = \emptyset$, for every $x \in \vec{v}$, a parameter $\gamma \in \{0,1\}$.
output: A structure $\mathcal{S}$ such that $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq [\![Q]\!]$, $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \emptyset$, and $|[\![Q']\!]|$
       and $|[\![\neg Q']\!]|$ contain at least $\min\{\left|\mathcal{T}^+_{\vec{v}}\right|, \left|\mathcal{T}^-_{\vec{v}}\right|\}$ tuples, for every $Q' \sqsubseteq Q$.
function $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma) =$
  switch $Q$ do
    case $r(t_1, \dots, t_{\iota(r)})$ do return $\{r^{\mathcal{S}} \mapsto \mathcal{T}^+_{\vec{v}}[t_1, \dots, t_{\iota(r)}]\}$;
    case $x \approx y$ do
      if there exist $d, d'$ such that $d \neq d'$ and $(d, d') \in \mathcal{T}^+_{\vec{v}}[x, y]$, or
        $d = d'$ and $(d, d') \in \mathcal{T}^-_{\vec{v}}[x, y]$ then fail;
    case $\neg Q'$ do return $\mathsf{dg}(Q', \vec{v}, \mathcal{T}^-_{\vec{v}}, \mathcal{T}^+_{\vec{v}}, \gamma)$;
    case $Q_1 \vee Q_2$ or $Q_1 \wedge Q_2$ do
      $(\mathcal{T}^1_{\vec{v}}, \mathcal{T}^2_{\vec{v}}) \leftarrow \{(\mathcal{T}^1_{\vec{v}}, \mathcal{T}^2_{\vec{v}}) \mid \left|\mathcal{T}^1_{\vec{v}}[x]\right| = \left|\mathcal{T}^2_{\vec{v}}[x]\right| = \left|\mathcal{T}^1_{\vec{v}}\right| = \left|\mathcal{T}^2_{\vec{v}}\right| = \min\{\left|\mathcal{T}^+_{\vec{v}}\right|, \left|\mathcal{T}^-_{\vec{v}}\right|\},$
        $\mathcal{T}^1_{\vec{v}}[x] \cap \mathcal{T}^2_{\vec{v}}[x] = \emptyset, (\mathcal{T}^1_{\vec{v}}[x] \cup \mathcal{T}^2_{\vec{v}}[x]) \cap (\mathcal{T}^+_{\vec{v}}[x] \cup \mathcal{T}^-_{\vec{v}}[x]) = \emptyset, \text{for all } x \in \vec{v}\}$;
      if $\gamma = 0$ then
        return $\mathsf{dg}(Q_1, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \gamma) \cup \mathsf{dg}(Q_2, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}}, \gamma)$;
      else
        switch $Q$ do
          case $Q_1 \vee Q_2$ do
            return $\mathsf{dg}(Q_1, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \gamma) \cup \mathsf{dg}(Q_2, \vec{v}, \mathcal{T}^1_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^+_{\vec{v}}, \gamma)$;
          case $Q_1 \wedge Q_2$ do
            return $\mathsf{dg}(Q_1, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^-_{\vec{v}}, \mathcal{T}^1_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \gamma) \cup \mathsf{dg}(Q_2, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}}, \gamma)$;
    case $\exists y.\, Q_y$ do
      $(\mathcal{T}^1_{\vec{v}\cdot y}, \mathcal{T}^2_{\vec{v}\cdot y}) \leftarrow \{(\mathcal{T}^1_{\vec{v}\cdot y}, \mathcal{T}^2_{\vec{v}\cdot y}) \mid \mathcal{T}^1_{\vec{v}\cdot y}[\vec{v}] = \mathcal{T}^+_{\vec{v}}, \mathcal{T}^2_{\vec{v}\cdot y}[\vec{v}] = \mathcal{T}^-_{\vec{v}},$
        $\left|\mathcal{T}^1_{\vec{v}\cdot y}[y]\right| = \left|\mathcal{T}^1_{\vec{v}\cdot y}\right| = \left|\mathcal{T}^+_{\vec{v}}\right|, \left|\mathcal{T}^2_{\vec{v}\cdot y}[y]\right| = \left|\mathcal{T}^2_{\vec{v}\cdot y}\right| = \left|\mathcal{T}^-_{\vec{v}}\right|, \mathcal{T}^1_{\vec{v}\cdot y}[y] \cap \mathcal{T}^2_{\vec{v}\cdot y}[y] = \emptyset\}$;
      return $\mathsf{dg}(Q_y, \vec{v} \cdot y, \mathcal{T}^1_{\vec{v}\cdot y}, \mathcal{T}^2_{\vec{v}\cdot y}, \gamma)$;
```

Formally, given a query Q and two sets of tuples $\mathcal{T}^+$ and $\mathcal{T}^-$ over a fixed domain $\mathcal{D}$, representing assignments of $\mathsf{fv}(Q)$, Data Golf produces a structure $\mathcal{S}$ (represented as a partial mapping from predicate symbols to their interpretations), such that $\mathcal{T}^+ \subseteq [\![Q]\!]$, $\mathcal{T}^- \cap [\![Q]\!] = \emptyset$, and $|[\![Q']\!]|$ and $|[\![\neg Q']\!]|$ contain at least $\min\{|\mathcal{T}^+|, |\mathcal{T}^-|\}$ tuples, for every $Q' \sqsubseteq Q$. To be able to produce such a structure $\mathcal{S}$, we make the following assumptions on Q:

CON the bound variable y in every subquery $\exists y.\, Q_y$ of Q satisfies $\mathsf{con}(y, Q_y, \mathcal{G})$ [14, Figure 5] for some set $\mathcal{G}$ such that $\mathsf{eqs}(y, \mathcal{G}) = \emptyset$ and, for every $Q_{qp} \in \mathcal{G}$, $\{y\} \subsetneq \mathsf{fv}(Q_{qp})$ holds; this avoids subqueries like $\exists y.\, \neg \mathsf{P}_2(x, y)$ and $\exists y.\, (\mathsf{P}_2(x, y) \vee \mathsf{P}_1(y))$;

CST Q contains no subquery of the form $x \approx \mathsf{c}$, which is satisfied by exactly one tuple;

VAR Q contains no closed subqueries, e.g., $\mathsf{P}_1(42)$, because a closed subquery is either satisfied by all possible tuples or no tuple at all; and

REP Q contains no repeated predicate symbols; this avoids subqueries like $\mathsf{P}_1(x) \wedge \neg \mathsf{P}_1(x)$.

Given a sequence of pairwise distinct variables $\vec{v}$ and a tuple $\vec{d}$ of the same length, we may interpret the tuple $\vec{d}$ as a *tuple over* $\vec{v}$, denoted as $\vec{d}(\vec{v})$. Given a sequence $t_1, \dots, t_k \in \vec{v} \cup \mathcal{C}$ of terms, we denote by $\vec{d}(\vec{v})[t_1, \dots, t_k]$ the tuple obtained by evaluating the terms $t_1, \dots, t_k$ over $\vec{d}(\vec{v})$. Formally, we define $\vec{d}(\vec{v})[t_1, \dots, t_k] \coloneqq (d'_i)_{i=1}^k$, where $d'_i = \vec{d_j}$ if $t_i = \vec{v}_j$ and $d'_i = t_i$ if $t_i \in \mathcal{C}$. We lift this notion to sets of tuples over $\vec{v}$ in the standard way.

Data Golf is formalized by the function $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$, defined in Algorithm 3, where $\vec{v}$ is a sequence of distinct variables such that $\mathsf{fv}(Q) \subseteq \vec{v}$, $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$ are sets of tuples over $\vec{v}$, and $\gamma \in \{0, 1\}$ is a *strategy*. The function $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$ can fail on an equality between two variables $x \approx y$. In this case, the function $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$ does not compute a Data Golf structure. We define the *not-depth* of a subquery $x \approx y$ in Q as the number of subqueries that have the form of a negation among the queries $x \approx y \sqsubseteq \cdots \sqsubseteq Q$, i.e., the number of negations on the path between the subquery $x \approx y$ and Q's main connective. To prevent failure, we generate the sets $\mathcal{T}^+_{\vec{v}}$, $\mathcal{T}^-_{\vec{v}}$ to only contain tuples with equal values for all variables in equalities with even (odd, respectively) not-depth and pairwise distinct values for all variables in equalities with odd (even, respectively) not-depth. This is not always possible, e.g., for $x \approx y \wedge \neg x \approx y$, in which case no Data Golf structure can be computed. In the case of a conjunction or a disjunction, we add disjoint sets $\mathcal{T}^1_{\vec{v}}$, $\mathcal{T}^2_{\vec{v}}$ of tuples over $\vec{v}$ to $\mathcal{T}^+_{\vec{v}}$, $\mathcal{T}^-_{\vec{v}}$ so that the intermediate results for the subqueries are neither equal nor disjoint. We implement two strategies (parameter γ) to choose these sets $\mathcal{T}^1_{\vec{v}}$, $\mathcal{T}^2_{\vec{v}}$.

Finally, we justify why a Data Golf structure $\mathcal{S}$ computed by $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$ satisfies $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq \llbracket Q \rrbracket$ and $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap \llbracket Q \rrbracket = \emptyset$. We proceed by induction on the query Q. Because of REP, the Data Golf structures for the subqueries Q_1, Q_2 of a binary query $Q_1 \vee Q_2$ or $Q_1 \wedge Q_2$ can be combined using the union operator. The only case that does not follow immediately is that $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap \llbracket Q \rrbracket = \emptyset$ for a query Q of the form $\exists y.\, Q_y$. We prove this case by contradiction. Without loss of generality we assume that $\vec{\mathsf{fv}}(Q_y) = \vec{\mathsf{fv}}(Q) \cdot y$. Suppose that $\vec{d} \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)]$ and $\vec{d} \in \llbracket Q \rrbracket$. Because $\vec{d} \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)]$, there exists some d such that $\vec{d} \cdot d \in \mathcal{T}^2_{\vec{v}\cdot y}[\vec{\mathsf{fv}}(Q_y)]$. Because $\vec{d} \in \llbracket Q \rrbracket$, there exists some d' such that $\vec{d} \cdot d' \in \llbracket Q_y \rrbracket$. By the induction hypothesis, $\vec{d} \cdot d \notin \llbracket Q_y \rrbracket$ and $\vec{d} \cdot d' \notin \mathcal{T}^2_{\vec{v}\cdot y}[\vec{\mathsf{fv}}(Q_y)]$. Because $\mathsf{con}(y, Q_y, \mathcal{G})$ holds for some $\mathcal{G}$ satisfying CON, the query Q_y is equivalent to $(Q_y \wedge \mathsf{qps}^{\vee}(\mathcal{G})) \vee Q_y[y/\bot]$. We have $\vec{d} \cdot d' \in \llbracket Q_y \rrbracket$. If the tuple $\vec{d} \cdot d'$ satisfies $Q_y[y/\bot]$, then $\vec{d} \cdot d \in \llbracket Q_y \rrbracket$ (contradiction) because the variable y does not occur in the query $Q_y[y/\bot]$ and thus its assignment in $\vec{d} \cdot d'$ can be arbitrarily changed. Otherwise, the tuple $\vec{d} \cdot d'$ satisfies some quantified predicate $Q_{qp} \in \mathsf{qps}(\mathcal{G})$ and (CON) implies $\{y\} \subsetneq \mathsf{fv}(Q_{qp})$. Hence, the tuples $\vec{d} \cdot d$ and $\vec{d} \cdot d'$ agree on the assignment of a variable $x \in \mathsf{fv}(Q_{qp}) \setminus \{y\}$. Let $\mathcal{T}^+_{\vec{v}'}$ and $\mathcal{T}^-_{\vec{v}'}$ be the sets in the recursive call of dg on the atomic predicate from Q_{qp}. Because $\vec{d} \cdot d \in \mathcal{T}^2_{\vec{v}\cdot y}[\vec{\mathsf{fv}}(Q_y)]$ and $\mathcal{T}^2_{\vec{v}\cdot y}[\vec{\mathsf{fv}}(Q_y)] \subseteq \mathcal{T}^+_{\vec{v}'}[\vec{\mathsf{fv}}(Q_y)] \cup \mathcal{T}^-_{\vec{v}'}[\vec{\mathsf{fv}}(Q_y)]$, the tuple $\vec{d} \cdot d$ is in $\mathcal{T}^+_{\vec{v}'}[\vec{\mathsf{fv}}(Q_y)] \cup \mathcal{T}^-_{\vec{v}'}[\vec{\mathsf{fv}}(Q_y)]$. Because $\vec{d} \cdot d'$ satisfies the quantified predicate Q_{qp}, the tuple $\vec{d} \cdot d'$ is in $\mathcal{T}^+_{\vec{v}'}[\vec{\mathsf{fv}}(Q_y)]$. Next we observe that the assignments of every variable (in particular, x) in the tuples from the sets $\mathcal{T}^+_{\vec{v}'}$, $\mathcal{T}^-_{\vec{v}'}$ are pairwise distinct (the conditions $\mathcal{T}^+_{\vec{v}'}[x] \cap \mathcal{T}^-_{\vec{v}'}[x] = \emptyset$, $\left|\mathcal{T}^+_{\vec{v}'}[x]\right| = \left|\mathcal{T}^+_{\vec{v}'}\right|$, and $\left|\mathcal{T}^-_{\vec{v}'}[x]\right| = \left|\mathcal{T}^-_{\vec{v}'}\right|$). Because the tuples $\vec{d} \cdot d$ and $\vec{d} \cdot d'$ agree on the assignment of x, they must be equal, i.e., $\vec{d} \cdot d = \vec{d} \cdot d'$ (contradiction).

The sets $\mathcal{T}^+_{\vec{v}}$, $\mathcal{T}^-_{\vec{v}}$ only grow in dg's recursion and the properties CON, CST, VAR, REP imply that Q has no closed subquery. Hence, $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq \llbracket Q \rrbracket$ and $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap \llbracket Q \rrbracket = \emptyset$ imply that $|\llbracket Q' \rrbracket|$ and $|\llbracket \neg Q' \rrbracket|$ contain at least $\min\{\left|\mathcal{T}^+_{\vec{v}}\right|, \left|\mathcal{T}^-_{\vec{v}}\right|\}$ tuples, for every $Q' \sqsubseteq Q$.

▶ **Example 18.** Consider the query $Q \coloneqq \neg\exists y.\, \mathsf{P}_2(x, y) \wedge \neg\mathsf{P}_3(x, y, z)$. This query Q satisfies the assumptions CON, CST, VAR, REP. In particular, $\mathsf{con}(y, \mathsf{P}_2(x, y) \wedge \neg\mathsf{P}_3(x, y, z), \mathcal{G})$ holds for $\mathcal{G} = \{\mathsf{P}_2(x, y)\}$ with $\{y\} \subsetneq \mathsf{fv}(\mathsf{P}_2(x, y))$. We choose $\vec{v} = (x, z)$, $\mathcal{T}^+_{\vec{v}} = \{(0, 4), (2, 6)\}$, and $\mathcal{T}^-_{\vec{v}} = \{(8, 12), (10, 14)\}$. The function $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$ first flips $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$ (because Q's main connective is negation) and then extends the tuples in the sets $\mathcal{T}^-_{\vec{v}}$ and $\mathcal{T}^+_{\vec{v}}$ with a value for the bound variable y: $\mathcal{T}^1_{\vec{v}\cdot y} = \{(8, 12, 16), (10, 14, 18)\}$ and $\mathcal{T}^2_{\vec{v}\cdot y} = \{(0, 4, 20), (2, 6, 22)\}$.

For conjunction (a binary operator), two additional sets of tuples are computed: $\overline{\mathcal{T}^1_{\vec{v}\cdot y}} = \{(24, 28, 32), (26, 30, 34)\}$ and $\overline{\mathcal{T}^2_{\vec{v}\cdot y}} = \{(36, 40, 44), (38, 42, 46)\}$. Depending on the strategy ($\gamma = 0$ or $\gamma = 1$), one of the following structures is computed: $\mathcal{S}_0 = \{\mathsf{P}_2 \mapsto \{(8, 16), (10, 18), (24, 32), (26, 34)\}, \mathsf{P}_3 \mapsto \mathcal{T}^+_{xyz}\}$, or $\mathcal{S}_1 = \{\mathsf{P}_2 \mapsto \{(8, 16), (10, 18), (0, 20), (2, 22)\}, \mathsf{P}_3 \mapsto \mathcal{T}^+_{xyz}\}$, where $\mathcal{T}^+_{xyz} = \{(0, 20, 4), (2, 22, 6), (24, 32, 28), (26, 34, 30)\}$.

The query $\mathsf{P}_1(x) \wedge Q$ is satisfied by the finite set of tuples $\mathcal{T}^+_{\vec{v}}$ under the structure $\mathcal{S}_1 \cup \{\mathsf{P}_1 \mapsto \{(0),(2)\}\}$ obtained by extending $\mathcal{S}_1$ ($\gamma = 1$). In contrast, the same query $\mathsf{P}_1(x) \wedge Q$ is satisfied by an infinite set of tuples including $\mathcal{T}^+_{\vec{v}}$ and disjoint from $\mathcal{T}^-_{\vec{v}}$ under the structure $\mathcal{S}_0 \cup \{\mathsf{P}_1 \mapsto \{(0),(2)\}\}$ obtained by extending $\mathcal{S}_0$ ($\gamma = 0$).

6 Implementation and Empirical Evaluation

We have implemented our translation RC2SQL consisting of roughly 1000 lines of OCaml code [25]. Although our translation satisfies the worst-case complexity bound (Theorem 14), we further improve its average-case complexity by implementing the following optimizations, described in more detail in our extended report [25, Section E].

- We use a sample structure of constant size, called a *training database*, to estimate the query cost when resolving the nondeterministic choices in our algorithms. A good training database should preserve the relative ordering of queries by their cost over the actual database as much as possible. Nevertheless, our translation satisfies the correctness and worst-case complexity claims (Section 4.3 and 4.4) for every choice of the training database. All our experiments used a Data Golf structure with $|\mathcal{T}^+| = |\mathcal{T}^-| = 2$ as the training database.
- We use the function optcnt optimizing RANF subqueries of the form $\exists \vec{y}.\, Q^+ \wedge \bigwedge_{i=1}^{k} \neg Q_i^-$ using the count aggregation operator. Inspired by Claußen et al. [9], we compare the number of assignments of $\vec{y}$ that satisfy Q^+ and $\bigvee_{i=1}^{k}(Q^+ \wedge Q_i^-)$, respectively.
- To compute an SQL query from a RANF query, we define the function $\mathsf{ranf2sql}(\cdot)$. We first obtain an equivalent RA expression using the standard approach [1] but adjusting the case of closed queries [8]. To translate RA expressions into SQL, we reuse a publicly available RA interpreter radb [30]. We modify its implementation to improve the performance of the resulting SQL query. We map the anti-join operator $\hat{Q}_1 \triangleright \hat{Q}_2$ to a more efficient LEFT JOIN, if $\mathsf{fv}(\hat{Q}_2) \subsetneq \mathsf{fv}(\hat{Q}_1)$, and we perform common subquery elimination.

To validate our translation's improved asymptotic time complexity, we compare it with the translation by Van Gelder and Topor [14] (VGT), an implementation of the algorithm by Ailamazyan et al. [2] that uses an extended active domain as the generators, and the DDD [20, 21], LDD [7], and MonPoly$^{\mathsf{REG}}$ [4] tools that support direct RC query evaluation using binary decision diagrams. We could not find a publicly available implementation of Van Gelder and Topor's translation. Therefore, the tool VGT for evaluable RC queries is derived from our implementation by modifying the function $\mathsf{rb}(\cdot)$ in Algorithm 1 to use the con relation [14, Figure 5] instead of $\mathsf{cov}(x, Q, \mathcal{G})$ (Figure 3) and to use the generator $\exists \vec{\mathsf{fv}}(Q) \setminus \{x\}.\, \mathsf{qps}^{\vee}(\mathcal{G})$ instead of $\mathsf{qps}^{\vee}(\mathcal{G})$. Evaluable queries Q are always translated into $(Q_{\mathit{fin}}, \bot)$ by $\mathsf{rw}(\cdot)$ because all of Q's free variables are range-restricted. We also consider translation variants that omit the count aggregation optimization $\mathsf{optcnt}(\cdot)$, marked with a minus ($^-$).

SQL queries computed by the translations are evaluated using the PostgreSQL database engine. We have also used the MySQL database engine but omit its timings from our evaluation after discovering that it computed incorrect results for some queries. This issue was reported and subsequently confirmed by MySQL developers. We run our experiments on an Intel Core i5-4200U CPU computer with 8 GB RAM. The relations in PostgreSQL are recreated before each invocation to prevent optimizations based on caching recent query evaluation results. We provide all our experiments in an easily reproducible artifact [25].

In the Small, Medium, and Large experiments, we generate ten pseudorandom queries with a fixed size 14 and Data Golf structures $\mathcal{S}$. The queries satisfy the Data Golf assumptions along with a few additional ones: the queries are not safe-range, have no repeated equalities, Experiment Small, Evaluable pseudorandom queries Q, $|\mathsf{sub}(Q)| = 14$, $n = 500$:

RC2SQL	**0.3**	0.3	**0.3**	**0.2**	**0.2**	**0.3**	**0.3**	**0.2**	**0.2**	0.3
RC2SQL$^-$	**0.3**	**0.2**	150.3	0.3	**0.2**	**0.3**	**0.3**	0.3	5.9	**0.2**
VGT	31.5	6.7	4.2	2.5	37.5	9.3	2.4	2.3	11.3	2.7
VGT$^-$	33.7	4.8	119.9	6.3	11.2	21.9	31.4	11.3	12.3	21.9
DDD	9.1	2.5	RE	7.1	5.9	RE	5.1	RE	2.2	5.1
LDD	59.2	24.1	169.1	38.8	53.3	37.4	64.0	TO	16.0	61.6
MonPoly$^{\mathsf{REG}}$	64.2	31.4	143.0	57.6	67.8	54.4	72.4	174.6	33.6	71.3

Experiment Medium, Evaluable pseudorandom queries Q, $|\mathsf{sub}(Q)| = 14$, $n = 20000$:

RC2SQL	2.6	1.4	**3.9**	2.1	**1.5**	2.8	3.3	**1.6**	**1.2**	2.6
RC2SQL$^-$	**2.0**	**1.0**	TO	**2.0**	1.7	**2.5**	**2.3**	1.8	TO	**1.8**
VGT	TO	TO	7.8	3.9	TO	TO	5.2	4.7	TO	4.8
VGT$^-$	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

Experiment Large, Evaluable pseudorandom queries Q, $|\mathsf{sub}(Q)| = 14$, tool = RC2SQL:

$n = 40000$	3.5	2.7	8.1	4.0	3.2	5.5	6.7	4.1	1.9	5.8
$n = 80000$	7.5	5.4	16.1	8.0	6.1	11.5	14.0	8.1	4.2	11.7
$n = 120000$	13.2	8.2	24.6	11.5	8.9	16.3	20.9	11.0	7.2	16.7

Experiment Infinite, Non-evaluable pseudorandom queries Q, $|\mathsf{sub}(Q)| = 7$, $n = 4000$:

	Infinite results ($\gamma = 0$)					Finite results ($\gamma = 1$)				
RC2SQL	0.8	0.8	0.8	0.8	0.8	1.0	1.1	0.9	**2.4**	**1.1**
RC2SQL$^-$	**0.5**	**0.5**	**0.4**	**0.5**	**0.5**	**0.6**	**0.7**	**0.6**	TO	2.0
DDD	89.5	49.1	46.9	116.3	50.4	81.7	44.1	45.8	89.8	44.6
LDD	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
MonPoly$^{\mathsf{REG}}$	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

Figure 4 Experiments Small, Medium, Large, and Infinite. We use the following abbreviations: TO = Timeout of 300s, RE = Runtime Error.

disjunction only appears at the top-level, every bound variable actually occurs in its scope, and only pairwise distinct variables appear as terms in predicates. The queries have 2 free variables and every subquery has at most 4 free variables. We control the size of the Data Golf structure $\mathcal{S}$ in our experiments using a parameter $n = |\mathcal{T}^+| = |\mathcal{T}^-|$. Because the sets $\mathcal{T}^+$ and $\mathcal{T}^-$ grow in the recursion on subqueries, relations in a Data Golf structure typically have more than n tuples. The values of the parameter n for Data Golf structures are summarized in Figure 4.

The Infinite experiment consists of five pseudorandom queries Q that are *not* evaluable and $\mathsf{rw}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$, where $Q_{\mathit{inf}} \neq \bot$. Specifically, the queries are of the form $Q_1 \wedge \forall x, y.\ Q_2 \longrightarrow Q_3$, where Q_1, Q_2, and Q_3 are either atomic predicates or equalities. For each query Q, we compare the performance of our tool to tools that directly evaluate Q on structures generated by the two Data Golf strategies (parameter γ), which trigger infinite or finite evaluation results on the considered queries. For infinite results, our tool outputs this fact (by evaluating Q_{inf}), whereas the other tools also output a finite representation of the infinite result. For finite results, all tools produce the same output.

Figure 4 shows the empirical evaluation results for the experiments Small, Medium, Large, and Infinite. All entries are execution times in seconds, TO is a timeout, and RE is a runtime error. Each column shows evaluation times for a unique pseudorandom query. The lowest time for a query is typeset in bold. We do not report the translation time because it does not contribute to the time complexity for a fixed query. Still, RC2SQL's translation time is at most 0.6 seconds on every query in our experiments. We also omit the rows for

Query	Q^{susp}		Q^{susp}_{user}		Q^{susp}_{text}	
Param. n	10^3	10^4	10^3	10^4	10^3	10^4
RC2SQL	**2.0**	**2.2**	**3.0**	**3.5**	**6.2**	**7.1**
RC2SQL$^-$	61.7	TO	63.4	TO	484.9	TO
VGT	3.9	2.9	–	–	213.2	TO
VGT$^-$	433.8	TO	–	–	495.4	TO
DDD	7.1	TO	6.3	TO	28.8	TO
LDD	36.3	TO	34.0	TO	213.9	TO
MonPoly$^{\mathsf{REG}}$	49.9	TO	47.3	TO	181.2	TO

Query	Q^{susp}		Q^{susp}_{user}		Q^{susp}_{text}	
Dataset	GC	MI	GC	MI	GC	MI
RC2SQL	**2.9**	**16.2**	**4.2**	**21.4**	**8.9**	**91.3**
RC2SQL$^-$	273.9	TO	270.1	TO	TO	TO
VGT	3.5	18.9	–	–	TO	TO
VGT$^-$	TO	TO	–	–	TO	TO
DDD	93.3	TO	90.1	TO	178.5	TO
LDD	TO	TO	TO	TO	TO	TO
MonPoly$^{\mathsf{REG}}$	TO	TO	TO	TO	TO	TO

Figure 5 Experiment with the queries Q^{susp}, Q^{susp}_{user}, and Q^{susp}_{text}. We use the following abbreviations: GC = Gift Cards dataset, MI = Musical Instruments dataset, TO = Timeout of 600s.

tools that time out or crash on all queries of an experiment, e.g., Ailamazyan et al. [2]. We conclude that our translation RC2SQL significantly outperforms all other tools on all queries and scales well to higher values of n, i.e., larger relations in the Data Golf structures, on all queries.

We also evaluate the tools on the queries Q^{susp} and Q^{susp}_{user} from the introduction and on the more challenging query $Q^{susp}_{text} := \mathsf{B}(b) \wedge \exists u, s, t.\ \forall p.\ \mathsf{P}(b, p) \longrightarrow \mathsf{S}(p, u, s) \vee \mathsf{T}(p, u, t)$ with an additional relation T that relates user's review text (variable t) to a product. The query Q^{susp}_{text} computes all brands for which there is a user, a score, and a review text such that all the brand's products were reviewed by that user with that score or by that user with that text. We use both Data Golf structures (strategy $\gamma = 1$) and real-world structures obtained from the Amazon review dataset [23]. The real-world relations P, S, and T are obtained by projecting the respective tables from the Amazon review dataset for some chosen product categories (abbreviated GC and MI in Figure 5) and the relation B contains all brands from P that have at least three products. Because the tool by Ailamazyan et al., DDD, LDD, and MonPoly$^{\mathsf{REG}}$ only support integer data, we injectively remap the string and floating-point values from the Amazon review dataset to integers.

Figure 5 shows the empirical evaluation results: execution times on Data Golf structures (left) and execution times on structures derived from the real-world dataset for two specific product categories (right). We remark that VGT cannot handle the query Q^{susp}_{user} as it is not evaluable [14]. Our translation RC2SQL significantly outperforms all other tools (except VGT on Q^{susp}, but RC2SQL still outperforms VGT) on both Data Golf and real-world structures. VGT$^-$ translates Q^{susp} into a RANF query with a higher query cost than RC2SQL$^-$. However, the optimization $\mathsf{optcnt}(\cdot)$ manages to rectify this inefficiency and thus VGT exhibits a comparable performance as RC2SQL. Specifically, the factor of 80× in query cost between VGT$^-$ and RC2SQL$^-$ improves to 1.1× in query cost between VGT and RC2SQL on a Data Golf structure with $n = 20$ [25]. Nevertheless, VGT does not finish evaluating the query Q^{susp}_{text} on GC and MI datasets within 10 minutes, unlike RC2SQL.

7 Conclusion

We presented a translation-based approach to evaluating arbitrary relational calculus queries over an infinite domain with improved time complexity over existing approaches. This contribution is an important milestone towards making the relational calculus a viable query language for practical databases. In future work, we plan to integrate into our base language features that database practitioners love, such as inequalities, bag semantics, or aggregations.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

2 Alfred K. Ailamazyan, Mikhail M. Gilula, Alexei P. Stolboushkin, and Grigorii F. Schwartz. Reduction of a relational model with infinite domains to the case of finite domains. *Doklady Akademii Nauk SSSR*, 286(2):308–311, 1986. URL: `http://mi.mathnet.ru/dan47310`.

3 Arnon Avron and Yoram Hirshfeld. On first order database query languages. In *LICS, July 15-18, 1991, Amsterdam, The Netherlands*, pages 226–231. IEEE Computer Society, 1991. `doi:10.1109/LICS.1991.151647`.

4 David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015. `doi:10.1145/2699444`.

5 Michael Benedikt and Leonid Libkin. Relational queries over interpreted structures. *J. ACM*, 47(4):644–680, 2000. `doi:10.1145/347476.347477`.

6 Achim Blumensath and Erich Grädel. Finite presentations of infinite structures: Automata and interpretations. *Theory Comput. Syst.*, 37(6):641–674, 2004. `doi:10.1007/s00224-004-1133-y`.

7 Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Decision diagrams for linear arithmetic. In *FMCAD, 15-18 November 2009, Austin, Texas, USA*, pages 53–60. IEEE, 2009. `doi:10.1109/FMCAD.2009.5351143`.

8 Jan Chomicki and David Toman. Implementing temporal integrity constraints using an active DBMS. *IEEE Trans. Knowl. Data Eng.*, 7(4):566–582, 1995. `doi:10.1109/69.404030`.

9 Jens Claußen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB, August 25-29, 1997, Athens, Greece*, pages 286–295. Morgan Kaufmann, 1997. URL: `http://www.vldb.org/conf/1997/P286.PDF`.

10 E. F. Codd. Relational completeness of data base sublanguages. *Research Report / RJ / IBM / San Jose, California*, RJ987, 1972.

11 Erling Ellingsen. Regex golf, 2013. URL: `https://alf.nu/RegexGolf`.

12 Martha Escobar-Molano, Richard Hull, and Dean Jacobs. Safety and translation of calculus queries with scalar functions. In Catriel Beeri, editor, *PODS, May 25-28, 1993, Washington, DC, USA*, pages 253–264. ACM Press, 1993. `doi:10.1145/153850.153909`.

13 Allen Van Gelder and Rodney W. Topor. Safety and correct translation of relational calculus formulas. In Moshe Y. Vardi, editor, *PODS, March 23-25, 1987, San Diego, California, USA*, pages 313–327. ACM, 1987. `doi:10.1145/28659.28693`.

14 Allen Van Gelder and Rodney W. Topor. Safety and translation of relational calculus queries. *ACM Trans. Database Syst.*, 16(2):235–278, 1991. `doi:10.1145/114325.103712`.

15 Richard Hull and Jianwen Su. Domain independence and the relational calculus. *Acta Informatica*, 31(6):513–524, 1994. `doi:10.1007/BF01213204`.

16 Michael Kifer. On safety, domain independence, and capturability of database queries (preliminary report). In Catriel Beeri, Joachim W. Schmidt, and Umeshwar Dayal, editors, *Proceedings of the Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness, June 28-30, 1988, Jerusalem, Israel*, pages 405–415. Morgan Kaufmann, 1988. `doi:10.1016/b978-1-4832-1313-2.50037-8`.

17 Nils Klarlund and Anders Møller. *MONA v1.4 User Manual.* BRICS, Department of Computer Science, University of Aarhus, January 2001. URL: `http://www.brics.dk/mona/`.

18 Leonid Libkin. *Elements of Finite Model Theory.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. `doi:10.1007/978-3-662-07003-1`.

19 Hong-Cheu Liu, Jeffrey Xu Yu, and Weifa Liang. Safety, domain independence and translation of complex value database queries. *Inf. Sci.*, 178(12):2507–2533, 2008. `doi:10.1016/j.ins.2008.02.005`.

20 Jesper B. Møller. DDDLIB: A library for solving quantified difference inequalities. In Andrei Voronkov, editor, *CADE, July 27-30, 2002, Copenhagen, Denmark*, volume 2392 of *Lecture Notes in Computer Science*, pages 129–133. Springer, 2002. doi:10.1007/3-540-45620-1_9.

21 Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Difference decision diagrams. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *CSL, September 20-25, 1999, Madrid, Spain*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 1999. doi:10.1007/3-540-48168-0_9.

22 Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013. doi:10.1145/2590989.2590991.

23 Jianmo Ni, Jiacheng Li, and Julian J. McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *EMNLP, November 3-7, 2019, Hong Kong, China*, pages 188–197. Association for Computational Linguistics, 2019. doi:10.18653/v1/D19-1018.

24 Robert A. Di Paola. The recursive unsolvability of the decision problem for the class of definite formulas. *J. ACM*, 16(2):324–327, 1969. doi:10.1145/321510.321524.

25 Martin Raszyk, David Basin, Srđan Krstić, and Dmitriy Traytel. Implementation, evaluation, and extended report associated with this paper, 2022. URL: https://github.com/rc2sql/rc2sql.

26 Peter Z. Revesz. *Introduction to Constraint Databases.* Texts in Computer Science. Springer, 2002. doi:10.1007/b97430.

27 Boris A Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70(4):569–572, 1950.

28 Moshe Y. Vardi. The decision problem for database dependencies. *Inf. Process. Lett.*, 12(5):251–254, 1981. doi:10.1016/0020-0190(81)90025-9.

29 Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *STOC, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982. doi:10.1145/800070.802186.

30 Jun Yang. radb, 2019. URL: https://github.com/junyang/radb.