EFFICIENT EVALUATION OF ARBITRARY RELATIONAL CALCULUS QUERIES

MARTIN RASZYK [a], DAVID BASIN [b], SRĐAN KRSTIĆ [b], AND DMITRIY TRAYTEL [c]

[a] DFINITY, Zurich, Switzerland
e-mail address: martin.raszyk@dfinity.org

[b] Department of Computer Science, ETH Zürich, Zurich, Switzerland
e-mail address: {basin, srdan.krstic}@inf.ethz.ch

[c] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
e-mail address: traytel@di.ku.dk

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.

1. Introduction

Codd's theorem states that all domain-independent queries of the relational calculus (RC) can be expressed in relational algebra (RA) [Cod72]. 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

Key words and phrases: Relational calculus, relative safety, safe range, query translation.

Preprint submitted to
Logical Methods in Computer Science

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

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

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 [AHV95]):

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

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 [Cod72], one could also use the active domain to obtain canonical, but far less efficient, generators.)

Van Gelder and Topor [GT87, GT91] 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:

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

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 [BL00, HS94, AGSS86] or to abandon finite relations entirely and evaluate queries using finite representations of infinite (but well-behaved) relations such as systems of constraints [Rev02] or automatic structures [BG04]. These approaches favor expressiveness over efficiency. But 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 [AH91]: 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.

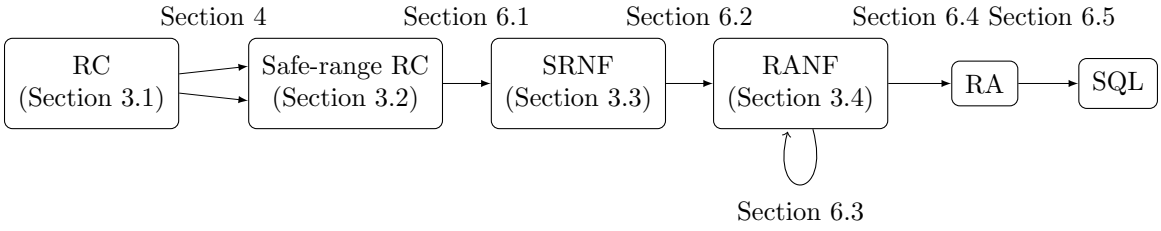

Figure 1: Overview of our translation.

Figure 1 summarizes our translation's steps and the sections where they are presented. Starting from an RC query, it produces two SQL queries via transformations to safe-range queries, the safe-range normal form (SRNF), the relational algebra normal form (RANF), and RA, respectively (Section 3). This article's main contribution is the first step: translating an RC query into two safe-range RC queries (Section 4), which fundamentally differs from Van Gelder and Topor's approach and produces better generators, like $\pi_{user,score}(\mathsf{S})$ above. Our generators strictly improve the time complexity of query evaluation (Section 5).

After the standard transformations from safe-range to RANF queries and from there to RA expressions, we translate the RA expressions into SQL using the radb tool [Yan19] (Section 6). We leverage various ideas from the literature to optimize the overall result. For example, we generalize Claußen et al. [CKMP97]'s approach to avoid evaluating Cartesian products like $\pi_{user,score}(\mathsf{S}) \times \mathsf{P}$ in RANF queries by using count aggregations (Section 6.3).

The translation to SQL enables any standard database management system (DBMS) to evaluate RC queries. We implement our translation and then use either PostgreSQL or MySQL for query evaluation. Using a real Amazon review dataset [NLM19] and our synthetic benchmark that generates hard database instances for random RC queries (Section 7), we evaluate our translation's performance (Section 8). The evaluation shows that our approach outperforms Van Gelder and Topor's translation (which also uses a standard DBMS for evaluation) and other RC evaluation approaches based on constraint databases and structure reduction.

In summary, our three main contributions are as follows:

- 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 [GT91].
- We implement our translation and extend it to produce SQL queries. The resulting tool RC2SQL makes RC a viable input language for any standard DBMS. 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.

This article extends our ICDT 2022 conference paper [RBKT22b] with a more complete description of the translation. In particular, it describes the steps that follow our main contribution – the translation of RC queries into two safe-range queries. In addition, we formally verify the functional correctness (but not the complexity analysis) of the main contribution using the Isabelle/HOL proof assistant [RT22]. The theorems and examples that have been verified in Isabelle are marked with a special symbol (). The formalization helped us identify and correct a technical oversight in the algorithm from the conference paper (even though the problem was compensated for by the subsequent steps of the translation in our implementation).

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 [Tra50] showed that it is undecidable whether there exists a (finite) structure and a variable assignment satisfying the query. In contrast, the question of whether a *fixed* structure and a *fixed* variable assignment satisfies the given RC query is decidable [AGSS86].

Kifer [Kif88] 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 [AH91] 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 [Var82] 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 [Lib04]. 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 [Cod72] affirmatively answers this question for domain-independent queries, restricting the structure to the *active domain*. Ailamazyan et al. [AGSS86] 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 [BL00] generalize Ailamazyan et al.'s [AGSS86] result to extensions of RC (e.g., with order relations) by combining the structure reduction with a translation-based approach. Hull and Su [HS94] 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 on finite tables. Van Gelder and Topor pioneered this approach [GT87, GT91] 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 [EHJ93, LYL08]. 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 (Appendix A). Our approach is inspired by Van Gelder and Topor's work 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 [Rev02] obviate the need for using RA operations on finite tables. This yields significant expressiveness gains as domain independence need not be assumed. Yet the efficiency of the quantifier elimination procedures employed cannot compare with the simple evaluation of the RA's projection operation. Similarly, automatic structures [BG04] 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 [MLAH99, Møl02, CGS09, KM01, BKMZ15].

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 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, writing $Q_1 \vee Q_2$ 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 as the (reflexive and transitive) subterm relation on the datatype of RC queries. For example, Q_1 is a subquery of the query $Q_1 \wedge \neg \exists y.\, Q_2$. We denote by $\mathsf{sub}(Q)$ the set of subqueries of a query Q, by $\mathsf{fv}(Q)$ the set of *free variables* in Q, and by $\mathsf{av}(Q)$ be the set of all (free and bound) variables in a query 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., $\mathsf{fv}(Q) = \varnothing$, 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., $\tilde{\exists} x.\, Q \coloneqq \exists x.\, Q$ if $x \in \mathsf{fv}(Q)$ and $\tilde{\exists} x.\, Q \coloneqq Q$ 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 [Lib04]) since the domain $\mathcal{D}$ can still be infinite. A (*variable*) *assignment* is a mapping $\alpha : \mathcal{V} \to \mathcal{D}$. We extend α to constant symbols $\mathsf{c} \in \mathcal{C}$ with $\alpha(\mathsf{c}) = \mathsf{c}^{\mathcal{S}}$. 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

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

Figure 2: The semantics of RC.

$$
\begin{array}{llllllllllll}
x \approx x & \equiv \top, & \neg\bot & \equiv \top, & \neg\top & \equiv \bot, & \exists x.\, \bot & \equiv \bot, & \exists x.\, \top & \equiv \top,\\
Q \wedge \bot & \equiv \bot, & \bot \wedge Q & \equiv \bot, & Q \wedge \top & \equiv Q, & \top \wedge Q & \equiv Q, & & \\
Q \vee \bot & \equiv Q, & \bot \vee Q & \equiv Q, & Q \vee \top & \equiv \top, & \top \vee Q & \equiv \top. & &
\end{array}
$$

Figure 3: Constant propagation rules.

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 2. 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 [Pao69, Var81].

We denote by $\mathsf{cp}(Q)$ the query obtained from a query Q by exhaustively applying the rules in Figure 3. Note that $\mathsf{cp}(Q)$ is either of the form $\bot$ or $\top$ or contains no $\bot$ or $\top$ subqueries.

Definition 3.1. *The substitution of the form $Q[x \mapsto y]$ is the query $\mathsf{cp}(Q')$, where Q' is obtained from a query Q by replacing all occurrences of the free variable x by the variable y, potentially also renaming bound variables to avoid capture.*

Definition 3.2. *The substitution of the form $Q[x/\bot]$ is the query $\mathsf{cp}(Q')$, where Q' is obtained from a query Q by replacing with $\bot$ every atomic predicate or equality containing the free variable x, except for $(x \approx x)$ which is replaced by $\top$.*

We lift the substitution notation to sets of queries in the standard way.

$\mathsf{gen}(x, \bot, \varnothing)$;
$\mathsf{gen}(x, Q, \{Q\})$ if $\mathsf{ap}(Q)$ and $x \in \mathsf{fv}(Q)$;
$\mathsf{gen}(x, \neg\neg Q, \mathcal{G})$ if $\mathsf{gen}(x, Q, \mathcal{G})$;
$\mathsf{gen}(x, \neg(Q_1 \vee Q_2), \mathcal{G})$
  if $\mathsf{gen}(x, (\neg Q_1) \wedge (\neg Q_2), \mathcal{G})$;
$\mathsf{gen}(x, \neg(Q_1 \wedge Q_2), \mathcal{G})$
  if $\mathsf{gen}(x, (\neg Q_1) \vee (\neg Q_2), \mathcal{G})$;
$\mathsf{gen}(x, Q_1 \vee Q_2, \mathcal{G}_1 \cup \mathcal{G}_2)$
  if $\mathsf{gen}(x, Q_1, \mathcal{G}_1)$ and $\mathsf{gen}(x, Q_2, \mathcal{G}_2)$;
$\mathsf{gen}(x, Q_1 \wedge Q_2, \mathcal{G})$
  if $\mathsf{gen}(x, Q_1, \mathcal{G})$ or $\mathsf{gen}(x, Q_2, \mathcal{G})$;
$\mathsf{gen}(x, Q \wedge x \approx y, \mathcal{G}[y \mapsto x])$
  if $\mathsf{gen}(y, Q, \mathcal{G})$;
$\mathsf{gen}(x, Q \wedge y \approx x, \mathcal{G}[y \mapsto x])$
  if $\mathsf{gen}(y, Q, \mathcal{G})$;
$\mathsf{gen}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G})$
  if $x \neq y$ and $\mathsf{gen}(x, Q_y, \mathcal{G})$.

Figure 4: The *generated* relation.

$\mathsf{cov}(x, x \approx x, \varnothing)$;
$\mathsf{cov}(x, Q, \varnothing)$ if $x \notin \mathsf{fv}(Q)$;
$\mathsf{cov}(x, x \approx y, \{x \approx y\})$ if $x \neq y$;
$\mathsf{cov}(x, y \approx x, \{x \approx y\})$ if $x \neq y$;
$\mathsf{cov}(x, Q, \{Q\})$ if $\mathsf{ap}(Q)$ and $x \in \mathsf{fv}(Q)$;
$\mathsf{cov}(x, \neg Q, \mathcal{G})$ if $\mathsf{cov}(x, Q, \mathcal{G})$;
$\mathsf{cov}(x, Q_1 \vee Q_2, \mathcal{G}_1 \cup \mathcal{G}_2)$ if $\mathsf{cov}(x, Q_1, \mathcal{G}_1)$ and $\mathsf{cov}(x, Q_2, \mathcal{G}_2)$;
$\mathsf{cov}(x, Q_1 \vee Q_2, \mathcal{G})$ if $\mathsf{cov}(x, Q_1, \mathcal{G})$ and $Q_1[x/\bot] = \top$;
$\mathsf{cov}(x, Q_1 \vee Q_2, \mathcal{G})$ if $\mathsf{cov}(x, Q_2, \mathcal{G})$ and $Q_2[x/\bot] = \top$;
$\mathsf{cov}(x, Q_1 \wedge Q_2, \mathcal{G}_1 \cup \mathcal{G}_2)$ if $\mathsf{cov}(x, Q_1, \mathcal{G}_1)$ and $\mathsf{cov}(x, Q_2, \mathcal{G}_2)$;
$\mathsf{cov}(x, Q_1 \wedge Q_2, \mathcal{G})$ if $\mathsf{cov}(x, Q_1, \mathcal{G})$ and $Q_1[x/\bot] = \bot$;
$\mathsf{cov}(x, Q_1 \wedge Q_2, \mathcal{G})$ if $\mathsf{cov}(x, Q_2, \mathcal{G})$ and $Q_2[x/\bot] = \bot$;
$\mathsf{cov}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G})$
  if $x \neq y$ and $\mathsf{cov}(x, Q_y, \mathcal{G})$ and $(x \approx y) \notin \mathcal{G}$;
$\mathsf{cov}(x, \exists y.\, Q_y, \tilde{\exists} y.\, (\mathcal{G} \setminus \{x \approx y\}) \cup \mathcal{G}_y[y \mapsto x])$
  if $x \neq y$ and $\mathsf{cov}(x, Q_y, \mathcal{G})$ and $\mathsf{gen}(y, Q_y, \mathcal{G}_y)$.

Figure 5: The *covered* relation.

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 [AHV95] is a decidable subset of domain-independent RC queries. Its definition is based on the notion of the 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" [AHV95]. 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 4. 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 $\mathcal{G}$. A similar definition by Van Gelder and Topor [GT91, Figure 5] uses a set of atomic (not quantified) predicates $\mathcal{A}$ as generators and defines the rule $\mathsf{gen}_{\mathsf{vgt}}(x, \exists y.\, Q_y, \mathcal{A})$ if $x \neq y$ and $\mathsf{gen}_{\mathsf{vgt}}(x, Q_y, \mathcal{A})$ (Appendix A, Figure 17). In contrast, we modify the rule's conclusion to existentially quantify the variable y in all queries in $\mathcal{G}$ where y is free: $\mathsf{gen}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G})$. Hence, $\mathsf{gen}(x, Q, \mathcal{G})$ implies $\mathsf{fv}(\mathcal{G}) \subseteq \mathsf{fv}(Q)$. We now formalize these relationships.

Lemma 3.3. *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 a $Q_{qp} \in \mathcal{G}$ such that $\alpha \models Q_{qp}$, and (iii) $Q[x/\bot] = \bot$.*

Definition 3.4. *Let $\mathsf{gen}(x, Q)$ hold iff $\mathsf{gen}(x, Q, \mathcal{G})$ holds for some $\mathcal{G}$. 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., $\mathsf{nongens}(Q) = \varnothing$. 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.*

$\mathsf{ranf}(\bot)$; $\mathsf{ranf}(\top)$;	
$\mathsf{ranf}(Q)$	if $\mathsf{ap}(Q)$;
$\mathsf{ranf}(\neg Q)$	if $\mathsf{ranf}(Q)$ and $\mathsf{fv}(Q) = \varnothing$;
$\mathsf{ranf}(Q_1 \vee Q_2)$	if $\mathsf{ranf}(Q_1)$ and $\mathsf{ranf}(Q_2)$ and $\mathsf{fv}(Q_1) = \mathsf{fv}(Q_2)$;
$\mathsf{ranf}(Q_1 \wedge Q_2)$	if $\mathsf{ranf}(Q_1)$ and $\mathsf{ranf}(Q_2)$;
$\mathsf{ranf}(Q_1 \wedge \neg Q_2)$	if $\mathsf{ranf}(Q_1)$ and $\mathsf{ranf}(Q_2)$ and $\mathsf{fv}(Q_2) \subseteq \mathsf{fv}(Q_1)$;
$\mathsf{ranf}(Q \wedge (x \approx y))$	if $\mathsf{ranf}(Q)$ and $\{x, y\} \cap \mathsf{fv}(Q) \neq \varnothing$;
$\mathsf{ranf}(Q \wedge \neg(x \approx y))$	if $\mathsf{ranf}(Q)$ and $\{x, y\} \subseteq \mathsf{fv}(Q)$;
$\mathsf{ranf}(\exists x.\, Q_x)$	if $\mathsf{ranf}(Q_x)$ and $x \in \mathsf{fv}(Q_x)$.

Figure 6: Characterization of RANF queries.

3.3. **Safe-Range Normal Form.** A query Q is in safe-range normal form (SRNF) if the query Q' in every subquery $\neg Q'$ of Q is an atomic predicate, equality, or an existentially quantified query [AHV95]. In Section 6.1 we define function $\mathsf{srnf}(Q)$ that returns a SRNF query equivalent to a query Q. Intuitively, the function $\mathsf{srnf}(Q)$ proceeds by pushing negations downwards [AHV95, Section 5.4], distributing existential quantifiers over disjunction [GT91, Rule (T9)], and dropping bound variables that never occur [GT91, Definition 9.2]. We include the last two rules to optimize the time complexity of evaluating the resulting query.

If a query Q is safe range, then $\mathsf{srnf}(Q)$ is also safe range.

3.4. **Relational Algebra Normal Form.** Relation algebra normal form (RANF) is a class of safe-range queries that can be easily mapped to RA[AHV95] and evaluated using the RA operations for projection, column duplication, selection, set union, binary join, and anti-join.

Figure 6 defines the predicate $\mathsf{ranf}(\cdot)$ characterizing RANF queries. The translation of safe-range queries (Section 3.2) to equivalent RANF queries proceeds via SRNF (Section 3.3). A safe-range query in SRNF can be translated to an equivalent RANF query by subquery rewriting using the following rules [AHV95, Algorithm 5.4.7]:

$$\begin{array}{lcll} Q \wedge (Q_1 \vee Q_2) & \equiv & (Q \wedge Q_1) \vee (Q \wedge Q_2), & (R1) \\ Q \wedge (\exists x.\, Q_x) & \equiv & (\exists x.\, Q \wedge Q_x), & (R2) \\ Q \wedge \neg Q' & \equiv & Q \wedge \neg(Q \wedge Q'). & (R3) \end{array}$$

Subquery rewriting is a nondeterministic process (as the rewrite rules can be applied in an arbitrary order) that impacts the performance of evaluating the resulting RANF query. We translate a safe-range query in SRNF to an equivalent RANF query by a recursive function $\mathsf{sr2ranf}(\cdot)$ inspired by the rules $(R1)$–$(R3)$ and fully specified in Figure 12 in Section 6.2.

3.5. **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, $\mathsf{cost}^{\mathcal{S}}(Q) \coloneqq \sum_{Q' \sqsubseteq Q,\ \mathsf{ranf}(Q')} \left|[\![Q']\!]^{\mathcal{S}}\right| \cdot |\mathsf{fv}(Q')|$. This corresponds to evaluating Q following its RANF structure (Section 3.4, Figure 6) 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_{fin}, Q_{inf}) such that

(FV) $\mathsf{fv}(Q_{fin}) = \mathsf{fv}(Q)$ unless Q_{fin} is syntactically equal to $\bot$; $\mathsf{fv}(Q_{inf}) = \varnothing$;
(EVAL) $[\![Q]\!]$ is an infinite set if Q_{inf} holds; otherwise $[\![Q]\!] = [\![Q_{fin}]\!]$ is a finite set.

Since the queries Q_{fin} and Q_{inf} are safe range, they are domain-independent and thus $[\![Q_{fin}]\!]$ is a finite set. In particular, $[\![Q]\!]$ is a finite set if Q_{inf} does not hold. Our translation generalizes Hull and Su's case distinction that restricts bound variables [HS94] 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 [GT91] while further improving the generators. Unless explicitly noted, in the rest of the article we assume a fixed structure $\mathcal{S}$.

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; x is range restricted in the first disjunct; the remaining disjuncts are all binary conjunctions of a query not containing x with a query of a special form containing x. The special form, central to our translation, is either an equality $x \approx y$ or a query satisfied by infinitely many values of x for all values of the remaining free variables.

We now restate Hull and Su's [HS94] and Van Gelder and Topor's [GT91] approaches using our notation in order to clarify how we generalize both approaches. In particular, Hull and Su's approach is already stated in a generalized way that restricts a *free* variable.

Hull and Su. Let $\tilde{Q}$ be a query with range-restricted bound variables and $x \in \mathsf{fv}(\tilde{Q})$. Then

$$\tilde{Q} \equiv \Big(\tilde{Q} \wedge \mathsf{AD}(x, \tilde{Q})\Big) \vee \Big(\bigvee\nolimits_{y \in \mathsf{fv}(\tilde{Q}) \setminus \{x\}} (\tilde{Q}[x \mapsto y] \wedge x \approx y)\Big) \vee \Big(\tilde{Q}[x/\bot] \wedge \neg(\mathsf{AD}(x, \tilde{Q}) \vee \bigvee\nolimits_{y \in \mathsf{fv}(\tilde{Q}) \setminus \{x\}} x \approx y)\Big).$$

Here $\mathsf{AD}(x, \tilde{Q})$ stands for an RC query with a single free variable x that is satisfied by an assignment α if and only if $\alpha(x) \in \mathsf{adom}(\tilde{Q})$. Hull and Su's translation distinguishes the following three cases for a fixed assignment α (each corresponding to a top-level disjunct above):

- if $\mathsf{AD}(x, \tilde{Q})$ holds (and hence $\alpha(x) \in \mathsf{adom}(\tilde{Q})$), 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 $\tilde{Q}[x/\bot]$. Specifically, all atomic predicates having x free can be replaced by $\bot$ (as $\alpha(x) \notin \mathsf{adom}(\tilde{Q})$), all equalities $x \approx y$ and $y \approx x$ for $y \in \mathsf{fv}(\tilde{Q}) \setminus \{x\}$ can be replaced by $\bot$ (as $\alpha(x) \neq \alpha(y)$), and all equalities $x \approx z$ for a bound variable z can be replaced by $\bot$ (as $\alpha(x) \notin \mathsf{adom}(\tilde{Q})$ and z is range restricted in its subquery $\exists z.\, Q_z$, by assumption). In the last case, $\mathsf{gen}(z, Q_z)$ holds and thus, for all α' extending α, we have $\alpha' \models \exists z.\, Q_z$ if and only if there exists a $d \in \mathsf{adom}(Q_z) \subseteq \mathsf{adom}(\tilde{Q})$ such that $\alpha'[z \mapsto d] \models Q_z$.

Example 4.1. *Consider the query* $Q \coloneqq \mathsf{B}(y) \vee x \approx y$. *Then* $\mathsf{AD}(x, Q) = \mathsf{B}(x)$ *and following Hull and Su we obtain that* Q *is equivalent to the disjunction of the following three queries:*

- $(\mathsf{B}(y) \vee x \approx y) \wedge \mathsf{B}(x)$, *which is equivalent to* $\mathsf{B}(x) \wedge \mathsf{B}(y)$;

- $(\mathsf{B}(y) \vee x \approx y)[x \mapsto y] \wedge x \approx y$, *which is syntactically equal to* $x \approx y$ *due to constant propagation that is part of the substitution operator;*
- $(\mathsf{B}(y) \vee x \approx y)[x/\bot] \wedge \neg(\mathsf{B}(x) \vee x \approx y)$, *which is equivalent to* $\mathsf{B}(y) \wedge \neg\mathsf{B}(x) \wedge \neg x \approx y$.

Note that in this example, each disjunct covers a different subset of Q*'s satisfying assignments and all three disjuncts are necessary to cover all of* Q*'s satisfying assignments.*

Van Gelder and Topor. Let $\tilde{Q}$ be an *evaluable* query with range-restricted bound variables, $x \in \mathsf{fv}(\tilde{Q})$. Then there exists a set $\mathcal{A}$ of atomic predicates such that

$$\tilde{Q} \equiv \left(\tilde{Q} \wedge \bigvee_{Q_{ap} \in \mathcal{A}} \exists \vec{\mathsf{fv}}(Q_{ap}) \setminus \{x\}.\, Q_{ap}\right) \vee \left(\tilde{Q}[x/\bot]\right).$$

Note that $\exists \vec{\mathsf{fv}}(Q) \setminus \{x\}.\, Q$ is the query in which all free variables of Q except for x are existentially quantified. Van Gelder and Topor restrict their attention to evaluable queries, which do not contain equalities between variables. (They only discuss an incomplete approach to supporting such equalities [GT91, Appendix A].) Thus, their translation lacks the corresponding disjuncts that Hull and Su have.

To avoid enumerating the entire active domain $\mathsf{adom}(\tilde{Q})$, Van Gelder and Topor replace the query $\mathsf{AD}(x, \tilde{Q})$ used by Hull and Su by the query $\bigvee_{Q_{ap} \in \mathcal{A}} \exists \vec{\mathsf{fv}}(Q_{ap}) \setminus \{x\}.\, Q_{ap}$ constructed from the atomic predicates from $\mathcal{A}$. Because their translation must yield an equivalent query (for every finite or infinite domain), $\mathcal{A}$ and $\tilde{Q}$ must satisfy, for all α,

$$\begin{array}{lll} \alpha \models \neg \bigvee_{Q_{ap} \in \mathcal{A}} \exists \vec{\mathsf{fv}}(Q_{ap}) \setminus \{x\}.\, Q_{ap} \implies (\alpha \models \tilde{Q} \iff \alpha \models \tilde{Q}[x/\bot]) & (\text{VGT}_1) & \text{and} \\ \alpha \models \tilde{Q}[x/\bot] \implies \alpha \models \forall x.\, \tilde{Q} & (\text{VGT}_2). & \end{array}$$

Note that (VGT_2) does not hold for the query $\tilde{Q} \coloneqq \neg\mathsf{B}(x)$ and thus Van Gelder and Topor only consider a proper subset of all RC queries, called evaluable. For evaluable queries, Van Gelder and Topor use the *constrained* relation $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{A})$, defined in Appendix A, Figure 17, to construct a set of atomic predicates $\mathcal{A}$ that satisfies (VGT_1).

Our Translation. Let $\tilde{Q}$ be a query with range-restricted bound variables, $x \in \mathsf{fv}(\tilde{Q})$. Then there exists a set $\mathcal{A}$ of atomic predicates and a set of equalities $\mathcal{E}$ such that

$$\begin{aligned} \tilde{Q} \equiv \left(\tilde{Q} \wedge \bigvee_{Q_{ap} \in \mathcal{A}} \exists \vec{\mathsf{fv}}(Q_{ap}) \setminus \mathsf{fv}(\tilde{Q}).\, Q_{ap}\right) \vee \left(\bigvee_{x \approx y \in \mathcal{E}} (\tilde{Q}[x \mapsto y] \wedge x \approx y)\right) \vee \\ \left(\tilde{Q}[x/\bot] \wedge \neg((\bigvee_{Q_{ap} \in \mathcal{A}} \exists \vec{\mathsf{fv}}(Q_{ap}) \setminus \mathsf{fv}(\tilde{Q}).\, Q_{ap}) \vee \bigvee_{x \approx y \in \mathcal{E}} x \approx y)\right). \end{aligned}$$

In contrast to Van Gelder and Topor, we only require that $\mathcal{A}$ satisfies (VGT_1) in our translation, which also allows us to translate non-evaluable queries, such as $\tilde{Q} := \neg\mathsf{B}(x)$ above. Note that we also existentially quantify only these variables that are not free in $\tilde{Q}$, whereas Van Gelder and Topor quantify all variables except x. For our introductory example Q^{susp}, this modification allows our translation to use the quantified predicate $\exists p.\, \mathsf{S}(p, u, s)$ to restrict both u and p simultaneously. In contrast, Van Gelder and Topor's approach restricts them separately using $\exists p, u.\, \mathsf{S}(p, u, s)$ and $\exists p, s.\, \mathsf{S}(p, u, s)$, so that the Cartesian product of these quantified predicates may need to be computed in their translated queries.

In contrast to Hull and Su, we do not consider the equalities of x with all other free variables in $\tilde{Q}$, but only such equalities $\mathcal{E}$ that occur in $\tilde{Q}$. We jointly compute the sets $\mathcal{A}$ and $\mathcal{E}$ using the *covered* relation $\mathsf{cov}(x, Q, \mathcal{G})$ (in contrast to $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{A})$ relation). Figure 5 shows the definition of this relation. The set $\mathcal{G}$ computed by the covered relation contains atomic predicates that satisfy (VGT_1) and are already quantified as described above. The set

also contains the relevant equalities that can be used in our translation. For every variable x and query $\tilde{Q}$ with range-restricted bound variables, there exists at least one set of quantified predicates and equalities $\mathcal{G}$ such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ and (VGT_1) holds for the set of atomic predicate subqueries in $\mathcal{G}$ (i.e., for $\{Q' \mid \mathsf{ap}(Q') \wedge \exists Q \in \mathcal{G}.\, Q' \sqsubseteq Q\}$). As the *cover* set $\mathcal{G}$ in $\mathsf{cov}(x, Q, \mathcal{G})$ may contain both quantified predicates and equalities between two variables, 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 $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ in the next lemma, which follows by induction on the derivation of $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$.

Lemma 4.2. *Let $\tilde{Q}$ be a query with range-restricted bound variables, $x \in \mathsf{fv}(\tilde{Q})$.*

Completeness: *Then there exists a set $\mathcal{G}$ of quantified predicates and equalities such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ holds and,*

Soundness: *for any $\mathcal{G}$ satisfying $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ and all α,*

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

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 $\exists \vec{\mathsf{fv}}(Q) \setminus \{x\}.\, \mathsf{qps}(\mathcal{G})$ as by Van Gelder and Topor. From Lemma 4.2, we derive our optimized translation characterized by the following lemma.

Lemma 4.3. *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*

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

Note that x is only guaranteed to be range restricted in (★)'s first disjunct. However, it only occurs in the remaining disjuncts in subqueries of a special form that are conjoined at the top-level to the disjuncts. These subqueries of a special form are equalities of the form $x \approx y$ or negations of a disjunction of quantified predicates with a free occurrence of x and equalities of the form $x \approx y$. We will show how to handle such occurrences in Section 4.2 and Section 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

input: An RC query Q.
output: A query $\tilde{Q}$ with range-restricted bound variables such that $Q \overset{\infty}{\equiv} \tilde{Q}$.

auxiliary function $\mathsf{fixbound}(\mathcal{Q}, x) =$
  $\{Q_{fix} \in \mathcal{Q} \mid x \in \mathsf{nongens}(Q_{fix})\}$
function $\mathsf{rb}(Q) =$
  switch Q **do**
    case $\neg Q'$ **do return** $\neg\mathsf{rb}(Q')$;
    case $Q'_1 \vee Q'_2$ **do return** $\mathsf{rb}(Q'_1) \vee \mathsf{rb}(Q'_2)$;
    case $Q'_1 \wedge Q'_2$ **do return** $\mathsf{rb}(Q'_1) \wedge \mathsf{rb}(Q'_2)$;
    case $\exists x.\, Q_x$ **do**
      $\mathcal{Q} \coloneqq \mathsf{flat}^{\vee}(\mathsf{rb}(Q_x))$;
      while $\mathsf{fixbound}(\mathcal{Q}, x) \neq \varnothing$ **do**
        $Q_{fix} \leftarrow \mathsf{fixbound}(\mathcal{Q}, x)$;
        $\mathcal{G} \leftarrow \{\mathcal{G} \mid \mathsf{cov}(x, Q_{fix}, \mathcal{G})\}$;
        $\mathcal{Q} \coloneqq (\mathcal{Q} \setminus \{Q_{fix}\}) \cup \{Q_{fix} \wedge \mathsf{qps}^{\vee}(\mathcal{G})\} \cup \bigcup_{y \in \mathsf{eqs}(x,\mathcal{G})} \{Q_{fix}[x \mapsto y]\} \cup \{Q_{fix}[x/\bot]\}$;
      return $\bigvee_{\tilde{Q} \in \mathcal{Q}} \tilde{\exists} x.\, \tilde{Q}$;
    otherwise do return Q;

Figure 7: Restricting bound variables.

input: An RC query Q.
output: Safe-range query pair (Q_{fin}, Q_{inf}) for which (FV) and (EVAL) hold.

auxiliary function $\mathsf{fixfree}(\mathcal{Q}_{fin}) =$
  $\{(Q_{fix}, E) \in \mathcal{Q}_{fin} \mid \mathsf{nongens}(Q_{fix}) \neq \varnothing\}$
auxiliary function $\mathsf{inf}(\mathcal{Q}_{fin}, Q) =$
  $\{(Q_f, E) \in \mathcal{Q}_{fin} \mid \mathsf{disjointvars}(Q_f, E) \neq \varnothing \vee \mathsf{fv}(Q_f) \cup \mathsf{fv}(E) \neq \mathsf{fv}(Q)\}$
function $\mathsf{split}(Q) =$
  $\mathcal{Q}_{fin} \coloneqq \{(\mathsf{rb}(Q), \varnothing)\}$; $\mathcal{Q}_{inf} \coloneqq \varnothing$;
  while $\mathsf{fixfree}(\mathcal{Q}_{fin}) \neq \varnothing$ **do**
    $(Q_{fix}, E) \leftarrow \mathsf{fixfree}(\mathcal{Q}_{fin})$;
    $x \leftarrow \mathsf{nongens}(Q_{fix})$;
    $\mathcal{G} \leftarrow \{\mathcal{G} \mid \mathsf{cov}(x, Q_{fix}, \mathcal{G})\}$;
    $\mathcal{Q}_{fin} \coloneqq (\mathcal{Q}_{fin} \setminus \{(Q_{fix}, E)\}) \cup \{(Q_{fix} \wedge \mathsf{qps}^{\vee}(\mathcal{G}), E)\} \cup \bigcup_{y \in \mathsf{eqs}(x,\mathcal{G})} \{(Q_{fix}[x \mapsto y], E \cup \{(x, y)\})\}$;
    $\mathcal{Q}_{inf} \coloneqq \mathcal{Q}_{inf} \cup \{Q_{fix}[x/\bot]\}$;
  while $\mathsf{inf}(\mathcal{Q}_{fin}, Q) \neq \varnothing$ **do**
    $(Q_f, E) \leftarrow \mathsf{inf}(\mathcal{Q}_{fin}, Q)$;
    $\mathcal{Q}_{fin} \coloneqq \mathcal{Q}_{fin} \setminus \{(Q_f, E)\}$;
    $\mathcal{Q}_{inf} \coloneqq \mathcal{Q}_{inf} \cup \{Q_f \wedge (\bigwedge_{Q \in \approx(E)} . Q)\}$;
  return $(\bigvee_{(Q_f,E) \in \mathcal{Q}_{fin}} (\bigwedge^{\approx}(Q_f, E)),$
    $\mathsf{rb}(\bigvee_{Q_i \in \mathcal{Q}_{inf}} \exists \vec{\mathsf{fv}}(Q_i).\, Q_i))$;

Figure 8: Restricting free variables.

the following inf-equivalence (recall: an equivalence that holds for infinite domains only):

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

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 4.4. *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*

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

Our approach for restricting all bound variables recursively applies Lemma 4.4. 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 Figure 7, 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 9 to establish the precondition of Lemma 4.4 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 (★∃) 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 (★∃) to Q_{fix} the variable x is either range restricted or does not occur in Q'_{fix}, i.e., $x \notin \mathsf{nongens}(Q'_{fix})$. This entails the termination of the loop on Lines 10–13.

Example 4.5. *Consider the query* $Q^{susp}_{user} \coloneqq \mathsf{B}(b) \wedge \exists s.\ \forall p.\ \mathsf{P}(b,p) \longrightarrow \mathsf{S}(p,u,s)$ *from Section 1. Restricting its bound variables yields the query*

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

The bound variable p is already range restricted in Q^{susp}_{user} and thus only s must be restricted. Applying (★) 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 quantifier over disjunction would yield the first disjunct in $\mathsf{rb}(Q^{susp}_{user})$ 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 $\mathsf{rb}(Q^{susp}_{user})$. This reasoning justifies that instead of (★) our algorithm applies (★∃) 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_{fin}, Q_{inf}) such that our translation's main properties (FV) and (EVAL) hold. Our translation is based on the following lemma.

Lemma 4.6. *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*

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

If $\tilde{Q}[x/\bot]$ is satisfied by some tuple, then $[\![\tilde{Q}]\!]$ 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 $[\![\tilde{Q}]\!]$ 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 4.6 (for y) implies that $[\![\tilde{Q}]\!]$ is an infinite set of tuples. Still, (☆) can be applied to $\tilde{Q}$ for x resulting in a query satisfied by the same infinite set of tuples.

Our approach is implemented by the function $\mathsf{split}(Q)$ defined in Figure 8. In the following, we describe this function and 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 $\mathcal{Q}_{fin}$ of pairs consisting of a query and a relation representing a set of equalities and a set $\mathcal{Q}_{inf}$ of queries such that

$$Q_{fin} \coloneqq \bigvee_{(Q_f,E)\in\mathcal{Q}_{fin}} \left(\bigwedge\nolimits^{\approx}(Q_f, E)\right), \qquad Q_{inf} \coloneqq \bigvee_{Q_i\in\mathcal{Q}_{inf}} \exists\vec{\mathsf{fv}}(Q_i).\, Q_i,$$

and, for every $(Q_f, E) \in \mathcal{Q}_{fin}$, the relation E represents a set of equalities between variables. Hereby, $\bigwedge^{\approx}(Q_f, E)$ is a query that is equivalent to $\bigwedge_{Q\in\{Q_f\}\cup\approx(E)} . Q$ where $\approx(E)$ abbreviates $\{x \approx y \mid (x, y) \in E\}$. However, the $\bigwedge^{\approx}(Q_f, E)$ operator carefully assembles the conjunction to ensure that the resulting query is safe range (whenever possible). In particular, the operator must iteratively conjoin the equalities from $\approx(E)$ to Q_f in a left-associative fashion and always pick next an equation for which one of the variables is free in Q_f or in the equalities conjoined so far, if such an equation exists. (If no such equation exists, the operator is free to conjoin the remaining equations in an arbitrary order.)

Our algorithm proceeds as follows. As long as there exists some $(Q_{fix}, E) \in \mathcal{Q}_{fin}$ such that $\mathsf{nongens}(Q_{fix}) \neq \varnothing$, we apply ($\bigstar$) to Q_{fix} and add the query $Q_{fix}[x/\bot]$ to $\mathcal{Q}_{inf}$. We remark that if we applied ($\bigstar$) to the entire disjunct $\bigwedge^{\approx}(Q_{fix}, E)$, the loop on Lines 7–12 might not terminate. Note that, for every (Q'_{fix}, E') added to $\mathcal{Q}_{fin}$ after applying ($\bigstar$) to Q_{fix}, $\mathsf{nongens}(Q'_{fix})$ is a proper subset of $\mathsf{nongens}(Q_{fix})$. This entails the termination of the loop on Lines 7–12. Finally, if $[\![Q_{fix}]\!]$ is an infinite set of tuples, then $\left[\!\left[\bigwedge^{\approx}(Q_{fix}, E)\right]\!\right]$ is an infinite set of tuples too. This is because the equalities in E merely duplicate columns of the query Q_{fix}. Hence, it indeed suffices to apply ($\bigstar$) to Q_{fix} instead of $\bigwedge^{\approx}(Q_{fix}, E)$.

After the loop on Lines 7–12 in Figure 8 terminates, for every $(Q_f, E) \in \mathcal{Q}_{fin}$, the query Q_f is safe range and E is a conjunction of equalities such that $\mathsf{fv}(Q_f) \cup \mathsf{fv}(E) \subseteq \mathsf{fv}(Q)$. However, the query $\bigwedge^{\approx}(Q_f, E)$ does not have to be safe range, e.g., if $Q_f \coloneqq \mathsf{B}(x)$ and $E \coloneqq \{(x, y), (u, v)\}$. Given a relation E, let $\mathsf{classes}(E)$ be the set of equivalence classes of free variables $\mathsf{fv}(\mathcal{Q}^{\approx})$ with respect to the (partial) equivalence closure of E, i.e., the smallest symmetric and transitive relation that contains E. For instance, $\mathsf{classes}(\{(x, y), (y, z), (u, v)\}) = \{\{x, y, z\}, \{u, v\}\}$. Let $\mathsf{disjointvars}(Q_f, E) \coloneqq \bigcup_{V\in\mathsf{classes}(E), V\cap\mathsf{fv}(Q_f)=\varnothing} V$ be the set of all variables in equivalence classes from $\mathsf{classes}(E)$ that are disjoint from Q_f's free variables. Then, $\bigwedge^{\approx}(Q_f, E)$ is safe range if and only if $\mathsf{disjointvars}(Q_f, E) = \varnothing$.[1]

Now if $\mathsf{disjointvars}(Q_f, E) \neq \varnothing$ and the query $\bigwedge^{\approx}(Q_f, E)$ is satisfied by some tuple, then $\left[\!\left[\bigwedge^{\approx}(Q_f, E)\right]\!\right]$ is an infinite set of tuples because all equivalence classes of variables in $\mathsf{disjointvars}(Q_f, E) \neq \varnothing$ can be assigned arbitrary values from the infinite domain $\mathcal{D}$. In our example with $Q_f \coloneqq \mathsf{B}(x)$ and $E \coloneqq \{(x, y), (u, v)\}$, we have $\mathsf{disjointvars}(Q_f, E) = \{u, v\} \neq \varnothing$. Moreover, if $\mathsf{fv}(Q_f) \cup \mathsf{fv}(E) \neq \mathsf{fv}(Q)$ and $\bigwedge^{\approx}(Q_f, E)$ 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 $\mathsf{fv}(Q) \setminus (\mathsf{fv}(Q_f) \cup \mathsf{fv}(E))$. Hence, for every $(Q_f, E) \in \mathcal{Q}_{fin}$ with $\mathsf{disjointvars}(Q_f, E) \neq \varnothing$ or $\mathsf{fv}(Q_f) \cup \mathsf{fv}(E) \neq \mathsf{fv}(Q)$, we remove (Q_f, E) from $\mathcal{Q}_{fin}$ and add $\bigwedge^{\approx}(Q_f, E)$ to $\mathcal{Q}_{inf}$. Note that we only remove pairs from $\mathcal{Q}_{fin}$, hence the loop on Lines 13–16 terminates. Afterwards, the query Q_{fin} is safe range. However, the query Q_{inf} does not have to be safe range. Indeed, every query $Q_i \in \mathcal{Q}_{inf}$ has

[1] This statement contained the error we discovered while formalizing the result presented in our conference paper [RBKT22b]. There we had wrongly used the naive conjunction $Q_f \wedge (\bigwedge_{Q\in\approx(E)} . Q)$, which will not be safe range whenever E has more than one element, instead of the more carefully constructed $\bigwedge^{\approx}(Q_f, E)$.

range-restricted bound variables, but not all the free variables of Q_i need be range restricted and thus the query $\exists \vec{\mathsf{fv}}(Q_i).\, Q_i$ does not have to 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 4.7. *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 $\mathsf{fv}(Q_{inf}) = \varnothing$.*

Lemma 4.8. *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 4.7, Q_{fin} is a safe-range (and thus also domain-independent) query. Hence, for the fixed structure, 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 4.8, if $\not\models Q_{inf}$, then $[\![Q_{fin}]\!]$ is equal to the "output-restricted unlimited interpretation" [HS94] 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 $Q \coloneqq \neg \mathsf{B}(x)$, 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 $[\![Q_{fin}]\!] = \varnothing$ is not equal to $[\![Q]\!] \cap \mathsf{adom}(Q)^{|\mathsf{fv}(Q)|}$.

Next, we demonstrate different aspects of our translation on a few examples. Thereby, we use a mildly modified algorithm that performs constant propagation after all steps that could introduce constants $\top$ or $\bot$ in a subquery. This optimization keeps the queries small, but is not necessary for termination and correctness. (In contrast, the constant propagation that is part of the substitution operators $Q[x \mapsto y]$ and $Q[x/\bot]$ is necessary.) We have verified in Isabelle that our results hold for the modified algorithm. That is, for all above theorems, we proved two variants: one with and one without additional constant propagation steps.

Example 4.9. *Consider the query $Q \coloneqq \mathsf{B}(x) \vee \mathsf{P}(x, y)$. 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 4.10. *Consider the query $Q \coloneqq \mathsf{B}(x) \wedge u \approx v$. 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 $Q_f = \mathsf{B}(x)$ and $E = \{(u, v)\}$. Now, all variables are range restricted in Q_f, but the variables in Q_f and E are disjoint. Hence, $[\![Q]\!]$ contains infinitely many tuples whenever Q_f is satisfied by some tuple. In contrast, $[\![Q]\!] = \varnothing$ if Q_f is never satisfied. Hence, we have $\mathsf{split}(Q) = (\bot, \exists x.\, \mathsf{B}(x))$.*

Example 4.11. *Consider the query $Q^{susp}_{user} \coloneqq \mathsf{B}(b) \wedge \exists s.\, \forall p.\, \mathsf{P}(b, p) \longrightarrow \mathsf{S}(p, u, s)$ 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 4.5. Splitting Q^{susp}_{user} yields*

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

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

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

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_{user}^{susp} *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,* $[\![Q_{user}^{susp}]\!]^{\mathcal{S}} = [\![\mathsf{rb}(Q_{user}^{susp})]\!]^{\mathcal{S}}$ *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_{user}^{susp} *is equivalent to* $\mathsf{rb}(Q_{user}^{susp}) \wedge (\exists s, p.\, \mathsf{S}(p,u,s))$ *obtained also by applying* (☆) *to* Q_{user}^{susp} *for the free variable* u*.*

5. Complexity Analysis

We analyze the time complexity of capturing Q, i.e., checking if $[\![Q]\!]$ is finite and enumerating $[\![Q]\!]$ in this case. To bound the asymptotic time complexity of capturing a fixed Q, we need to apply an additional standard translation step to both queries produced by our translation to obtain two RANF queries. Query cost (Section 3.5) can then be applied to the resulting two queries to bound computation time based on the cardinalities of subquery evaluation results.

Definition 5.1. *Let* Q *be an RC query and* $\mathsf{split}(Q) = (Q_{fin}, Q_{inf})$*. Let* $\hat{Q}_{fin} \coloneqq \mathsf{sr2ranf}(Q_{fin})$ *and* $\hat{Q}_{inf} \coloneqq \mathsf{sr2ranf}(Q_{inf})$ *be the equivalent RANF queries. We define* $\mathsf{rw}(Q) \coloneqq (\hat{Q}_{fin}, \hat{Q}_{inf})$*.*

Since function $\mathsf{sr2ranf}(\cdot)$ is a standard translation step, we present it in Section 6 (see Figure 12). Note that the proof of Lemma 5.6 relies on its algorithmic details.

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. Still, the RANF queries $\hat{Q}_{fin}$ and $\hat{Q}_{inf}$ computed by our translation need not have pairwise distinct (free and bound) variables.

We define the relation $\lesssim_Q$ on $\mathsf{av}(Q)$ such that $x \lesssim_Q y$ 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 $x \lesssim_Q y$ 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 $\lesssim_Q$ 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 $x \lesssim_Q y$, 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 $x \lesssim_Q y$;
- $\exists x.\, Q_{qp} \in \overline{\mathsf{qps}}(Q)$ if $Q_{qp} \in \overline{\mathsf{qps}}(Q)$, $x \in \mathsf{fv}(Q_{qp}) \setminus \mathsf{fv}(Q)$, and $x \lesssim_Q y$ for all $y \in \mathsf{fv}(Q_{qp})$.

We bound the time 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 $\mathsf{minimal}(\mathcal{Q}_{qps}) \coloneqq \forall Q_{qp} \in \mathcal{Q}_{qps}.\, \mathsf{fv}(\mathcal{Q}_{qps} \setminus \{Q_{qp}\}) \neq \mathsf{fv}(\mathcal{Q}_{qps})$. 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. [NRR13], we use the notation $\tilde{\mathcal{O}}(\cdot)$ to hide logarithmic factors incurred by set operations.

Theorem 5.2. *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 $[\![Q]\!]$ in this case, is in $\tilde{\mathcal{O}}\left(\sum_{\mathcal{Q}_{qps} \subseteq \overline{\mathsf{qps}}(Q), \mathsf{minimal}(\mathcal{Q}_{qps})} \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]|\right)$.*

Before we prove Theorem 5.2 we first provide some examples to reinforce the intuition behind our claim. Examples 5.3 and 5.4 show that the time complexity from Theorem 5.2 cannot be achieved by the translation of Van Gelder and Topor [GT91] or over finite domains. Example 5.5 shows how equalities affect the bound in Theorem 5.2.

Example 5.3. *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* [GT91]

$$\Big(\big(\exists s, p.\, \mathsf{S}(p,u,s)\big) \wedge \big(\exists u, p.\, \mathsf{S}(p,u,s)\big) \wedge \mathsf{B}(b)\Big) \wedge \\ \neg\exists p.\, \Big(\big(\exists s, p.\, \mathsf{S}(p,u,s)\big) \wedge \big(\exists u, p.\, \mathsf{S}(p,u,s)\big) \wedge \mathsf{P}(b,p)\Big) \wedge \neg\mathsf{S}(p,u,s)$$

restricts the variables u and s by $\exists s, p.\, \mathsf{S}(p,u,s)$ and $\exists u, p.\, \mathsf{S}(p,u,s)$, respectively. Note that this corresponds to the RA expression shown in Section 1 with the highlighted generators replaced with $\pi_{user}(\mathsf{S}) \times \pi_{score}(\mathsf{S})$.

Consider an interpretation of B by $\{(\mathsf{c}') \mid \mathsf{c}' \in \{1, \ldots, n\}\}$, P by $\{(\mathsf{c}', \mathsf{c}') \mid \mathsf{c}' \in \{1, \ldots, n\}\}$, and S by $\{(\mathsf{c}, \mathsf{c}', \mathsf{c}') \mid \mathsf{c} \in \{1, \ldots, n\}, \mathsf{c}' \in \{1, \ldots, m\}\}$, $n, m \in \mathbb{N}$. 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 5.2 yields an asymptotically better time complexity in $\tilde{\mathcal{O}}(n + m + n \cdot m)$ for our translation, more precisely:

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

which corresponds to the complexity of evaluating the RA expression shown in Section 1.

Example 5.4. *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 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 $\tilde{\mathcal{O}}(|[\![\mathsf{S}(x,y,z)]\!]|)$ of Theorem 5.2. Hence, our infinite domain assumption is crucial for achieving the better complexity bound.

Example 5.5. *Consider the following query over the infinite domain $\mathcal{D} = \mathbb{N}$ of natural numbers:*

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

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 5.2. 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 5.2 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* $Q \equiv \mathsf{B}(x) \wedge \mathsf{B}(y) \wedge \mathsf{B}(z)$*.*

Now, to prove Theorem 5.2, we need to introduce guard queries and the set of quantified predicates of a query. Given a RANF query $\hat{Q}$, we define a *guard* query $\mathsf{guard}(\hat{Q})$ that is implied by $\hat{Q}$, i.e., $\mathsf{guard}(\hat{Q})$ can be used to over-approximate the set of satisfying tuples for $\hat{Q}$. We use this over-approximation in our proof of Theorem 5.2. The guard query $\mathsf{guard}(\hat{Q})$ has a simple structure: it is the disjunction of conjunctions of quantified predicates and equalities.

We now define the set of quantified predicates $\mathsf{qps}(Q)$ occurring in the guard query $\mathsf{guard}(Q)$. For an atomic predicate $Q_{ap} \in \mathsf{aps}(Q)$, let $\mathcal{B}_Q(Q_{ap})$ be the set of sequences of bound variables for all occurrences of Q_{ap} in Q. For example, consider a query $Q_{ex} \coloneqq ((\exists z.\, (\exists y, z.\, \mathsf{P}_3(x, y, z)) \wedge \mathsf{P}_2(y, z)) \wedge \mathsf{P}_1(z)) \vee \mathsf{P}_3(x, y, z)$. Then $\mathsf{aps}(Q_{ex}) = \{\mathsf{P}_1(z), \mathsf{P}_2(y, z), \mathsf{P}_3(x, y, z)\}$ and $\mathcal{B}_{Q_{ex}}(\mathsf{P}_3(x, y, z)) = \{[y, z], []\}$, where [] denotes the empty sequence corresponding to the occurrence of $\mathsf{P}_3(x, y, z)$ in Q_{ex} for which the variables x, y, z are all free in Q_{ex}. Note that the variable z in the other occurrence of $\mathsf{P}_3(x, y, z)$ in Q_{ex} is bound to the innermost quantifier. Hence, neither $[z, y]$ nor $[z, y, z]$ are in $\mathcal{B}_{Q_{ex}}(\mathsf{P}_3(x, y, z))$. Furthermore, let $\mathsf{qps}(Q)$ be the set of the quantified predicates obtained by existentially quantifying sequences of bound variables in $\mathcal{B}_{Q'}(Q_{ap})$ from the atomic predicates $Q_{ap} \in \mathsf{aps}(Q')$ in all subqueries Q' of Q. Formally, $\mathsf{qps}(Q) \coloneqq \bigcup_{Q' \sqsubseteq Q, Q_{ap} \in \mathsf{aps}(Q')} \{\exists \vec{v}.\, Q_{ap} \mid \vec{v} \in \mathcal{B}_{Q'}(Q_{ap})\}$. For instance, $\mathsf{qps}(Q_{ex}) = \{\mathsf{P}_3(x, y, z), \exists z.\, \mathsf{P}_3(x, y, z), \exists y, z.\, \mathsf{P}_3(x, y, z), \mathsf{P}_2(y, z), \exists z.\, \mathsf{P}_2(y, z), \mathsf{P}_1(z)\}$.

A crucial property of our translation, which is central for the proof of Theorem 5.2, is the relationship between the quantified predicates $\mathsf{qps}(\hat{Q})$ for a RANF query $\hat{Q}$ produced by our translation and the original query Q. The relationship is formalized in the following lemma.

Lemma 5.6. *Let* Q *be an RC query with pairwise distinct (free and bound) variables and let* $\mathsf{rw}(Q) = (\hat{Q}_{fin}, \hat{Q}_{inf})$*. Let* $\hat{Q} \in \{\hat{Q}_{fin}, \hat{Q}_{inf}\}$*. Then* $\mathsf{qps}(\hat{Q}) \subseteq \overline{\mathsf{qps}}(Q)$*.*

Proof. Let $\mathsf{split}(Q) = (Q_{fin}, Q_{inf})$. We observe that $\mathsf{aps}(Q_{fin}) \subseteq \overline{\mathsf{qps}}(Q)$, $\mathsf{eqs}^*(Q_{fin}) \subseteq \mathsf{eqs}^*(Q)$, $\lesssim_{Q_{fin}} \subseteq \lesssim_Q$, $\mathsf{aps}(Q_{inf}) \subseteq \overline{\mathsf{qps}}(Q)$, $\mathsf{eqs}^*(Q_{inf}) \subseteq \mathsf{eqs}^*(Q)$, and $\lesssim_{Q_{inf}} \subseteq \lesssim_Q$. Hence, $\overline{\mathsf{qps}}(Q_{fin}) \subseteq \overline{\mathsf{qps}}(Q)$ and $\overline{\mathsf{qps}}(Q_{inf}) \subseteq \overline{\mathsf{qps}}(Q)$.

Next we observe that $\mathsf{qps}(Q') \subseteq \overline{\mathsf{qps}}(Q')$ for every query Q'. Finally, we show that $\mathsf{qps}(\hat{Q}_{fin}) \subseteq \mathsf{qps}(Q_{fin})$ and $\mathsf{qps}(\hat{Q}_{inf}) \subseteq \mathsf{qps}(Q_{inf})$. We observe that $\mathcal{B}_{\mathsf{cp}(Q')}(Q_{ap}) \subseteq \mathcal{B}_{Q'}(Q_{ap})$, $\mathcal{B}_{\mathsf{srnf}(Q')}(Q_{ap}) \subseteq \mathcal{B}_{Q'}(Q_{ap})$, and then $\mathsf{qps}(\mathsf{cp}(Q')) \subseteq \mathsf{qps}(Q')$, $\mathsf{qps}(\mathsf{srnf}(Q')) \subseteq \mathsf{qps}(Q')$, for every query Q'.

Assume that $Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ is a safe-range query in which no variable occurs both free and bound, no bound variables shadow each other, i.e., there are no subqueries $\exists x.\, Q_x \sqsubseteq Q'_x$ and $\exists x.\, Q'_x \sqsubseteq Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$, and every two subqueries $\exists x.\, Q_x \sqsubseteq Q_1$ and $\exists x.\, Q'_x \sqsubseteq Q_2$ such that $Q_1 \wedge Q_2 \sqsubseteq Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ have the property that $\exists x.\, Q_x$ or $\exists x.\, Q'_x$ is a quantified predicate. Then the free variables in $\bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ never clash with the bound variables in Q', i.e., Line 26 in Figure 12 is never executed. Next we observe that $\mathcal{B}_{\mathsf{sr2ranf}(Q', \mathcal{Q})}(Q_{ap}) \subseteq \mathcal{B}_{Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}}(Q_{ap})$ (this subset relation only holds when considering queries modulo α-equivalence, i.e., queries that have the same binding structure but differ in the used bound variable names are considered to be equal) and then $\mathsf{qps}(\mathsf{sr2ranf}(Q', \mathcal{Q})) \subseteq \mathsf{qps}(Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q})$. Because Q_{fin}, Q_{inf} have the assumed properties and $\mathsf{qps}(\mathsf{srnf}(Q')) \subseteq \mathsf{qps}(Q')$, for every query Q', we get $\mathsf{qps}(\hat{Q}_{fin}) = \mathsf{qps}(\mathsf{sr2ranf}(Q_{fin})) \subseteq \mathsf{qps}(Q_{fin})$ and $\mathsf{qps}(\hat{Q}_{inf}) = \mathsf{qps}(\mathsf{sr2ranf}(Q_{inf})) \subseteq \mathsf{qps}(Q_{inf})$. □

Recall Example 5.3. The query $\exists u, p.\, \mathsf{S}(p, u, s)$ is in $\mathsf{qps}(Q_{vgt})$, but not in $\overline{\mathsf{qps}}(Q)$. Hence, $\mathsf{qps}(Q_{vgt}) \subseteq \overline{\mathsf{qps}}(Q)$, i.e., an analogue of Lemma 5.6 for Van Gelder and Topor's translation, does not hold.

Every tuple satisfying a RANF query $\hat{Q}$ belongs to the set of tuples satisfying the join over some minimal subset $\mathcal{Q}_{qps} \subseteq \mathsf{qps}(\hat{Q})$ of quantified predicates and satisfying equalities duplicating some of $\mathcal{Q}_{qps}$'s columns. Hence, we define the guard query $\mathsf{guard}(\hat{Q})$ as follows:

$$\mathsf{guard}(\hat{Q}) \coloneqq \bigvee_{\substack{\mathcal{Q}_{qps} \subseteq \mathsf{qps}(\hat{Q}), \mathsf{minimal}(\mathcal{Q}_{qps}),\\ \mathcal{Q}^{\approx} \subseteq \{x \approx y \mid x \in \mathsf{fv}(\mathcal{Q}_{qps}) \wedge y \in \mathsf{fv}(\hat{Q})\},\\ \mathsf{fv}(\mathcal{Q}_{qps}) \cup \mathsf{fv}(\mathcal{Q}^{\approx}) = \mathsf{fv}(\hat{Q})}} \left(\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp} \wedge \bigwedge_{Q^{\approx} \in \mathcal{Q}^{\approx}} Q^{\approx} \right).$$

Note that $\{x \approx y \mid x \in V \wedge y \in V'\}$ denotes the set of all equalities $x \approx y$ between variables $x \in V$ and $y \in V'$. We express the correctness of the guard query in the following lemma.

Lemma 5.7. *Let $\hat{Q}$ be a RANF query. Then, for all variable assignments α,*

$$\alpha \models \hat{Q} \Longrightarrow \alpha \models \mathsf{guard}(\hat{Q}).$$

Moreover, $\mathsf{fv}(\mathsf{guard}(\hat{Q})) = \mathsf{fv}(\hat{Q})$ *unless* $\mathsf{guard}(\hat{Q}) = \bot$. *Hence,* $[\![\hat{Q}]\!]$ *satisfies*

$$\left[\!\!\left[\hat{Q}\right]\!\!\right] \subseteq \bigcup_{\substack{\mathcal{Q}_{qps} \subseteq \mathsf{qps}(\hat{Q}), \mathsf{minimal}(\mathcal{Q}_{qps}),\\ \mathcal{Q}^{\approx} \subseteq \{x \approx y \mid x \in \mathsf{fv}(\mathcal{Q}_{qps}) \wedge y \in \mathsf{fv}(\hat{Q})\},\\ \mathsf{fv}(\mathcal{Q}_{qps}) \cup \mathsf{fv}(\mathcal{Q}^{\approx}) = \mathsf{fv}(\hat{Q})}} \left[\!\!\left[\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp} \wedge \bigwedge_{Q^{\approx} \in \mathcal{Q}^{\approx}} Q^{\approx} \right]\!\!\right].$$

Proof. The statement follows by well-founded induction over the definition of $\mathsf{ranf}(\hat{Q})$. □

We now derive a bound on $|[\![\hat{Q}']\!]|$, for an arbitrary RANF subquery $\hat{Q}' \sqsubseteq \hat{Q}$, $\hat{Q} \in \{\hat{Q}_{fin}, \hat{Q}_{inf}\}$.

Lemma 5.8. *Let Q be an RC query with pairwise distinct (free and bound) variables and let $\mathsf{rw}(Q) = (\hat{Q}_{fin}, \hat{Q}_{inf})$. Let $\hat{Q}' \sqsubseteq \hat{Q}$ be a RANF subquery of $\hat{Q} \in \{\hat{Q}_{fin}, \hat{Q}_{inf}\}$. Then*

$$\left| \left[\!\!\left[\hat{Q}'\right]\!\!\right] \right| \leq \textstyle\sum_{\mathcal{Q}_{qps} \subseteq \overline{\mathsf{qps}}(Q), \mathsf{minimal}(\mathcal{Q}_{qps})} 2^{|\mathsf{av}(\hat{Q})|} \cdot \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]| \,.$$

Proof. Applying Lemma 5.7 to the RANF query $\hat{Q}'$ yields

$$\left[\!\!\left[\hat{Q}'\right]\!\!\right] \subseteq \bigcup_{\substack{\mathcal{Q}_{qps} \subseteq \mathsf{qps}(\hat{Q}'), \mathsf{minimal}(\mathcal{Q}_{qps}),\\ \mathcal{Q}^{\approx} \subseteq \{x \approx y \mid x \in \mathsf{fv}(\mathcal{Q}_{qps}) \wedge y \in \mathsf{fv}(\hat{Q}')\},\\ \mathsf{fv}(\mathcal{Q}_{qps}) \cup \mathsf{fv}(\mathcal{Q}^{\approx}) = \mathsf{fv}(\hat{Q}')}} \left[\!\!\left[\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp} \wedge \bigwedge_{Q^{\approx} \in \mathcal{Q}^{\approx}} Q^{\approx} \right]\!\!\right].$$

We observe that $|[\![\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp} \wedge \bigwedge_{Q^{\approx} \in \mathcal{Q}^{\approx}} Q^{\approx}]\!]| \leq |[\![\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp}]\!]| \leq \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]|$ where the first inequality follows from the fact that equalities $Q^{\approx} \in \mathcal{Q}^{\approx}$ can only restrict a set of tuples and duplicate columns. Because $\hat{Q}'$ is a subquery of $\hat{Q}$, it follows that $\mathsf{qps}(\hat{Q}') \subseteq \mathsf{qps}(\hat{Q})$. Lemma 5.6 yields $\mathsf{qps}(\hat{Q}) \subseteq \overline{\mathsf{qps}}(Q)$. Hence, we derive $\mathsf{qps}(\hat{Q}') \subseteq \overline{\mathsf{qps}}(Q)$.

The number of equalities in $\{x \approx y \mid x \in \mathsf{fv}(\mathcal{Q}_{qps}) \wedge y \in \mathsf{fv}(\hat{Q}')\}$ is at most

$$|\mathsf{fv}(\mathcal{Q}_{qps})| \cdot \left|\mathsf{fv}(\hat{Q}')\right| \leq \left|\mathsf{fv}(\hat{Q}')\right|^2 \leq \left|\mathsf{av}(\hat{Q})\right|^2.$$

The first inequality holds because $\mathsf{fv}(\mathcal{Q}_{qps}) \cup \mathsf{fv}(\mathcal{Q}^{\approx}) = \mathsf{fv}(\hat{Q}')$ and thus $\mathsf{fv}(\mathcal{Q}_{qps}) \subseteq \mathsf{fv}(\hat{Q}')$. The second inequality holds because the variables in a subquery $\hat{Q}'$ of $\hat{Q}$ are in $\mathsf{av}(\hat{Q})$. Hence, the number of subsets $\mathcal{Q}^{\approx} \subseteq \{x \approx y \mid x \in \mathsf{fv}(\mathcal{Q}_{qps}) \wedge y \in \mathsf{fv}(\hat{Q}')\}$ is at most $2^{\left|\mathsf{av}(\hat{Q})\right|^2}$. □

We now bound the query cost of a RANF query $\hat{Q} \in \{\hat{Q}_{fin}, \hat{Q}_{inf}\}$ over the fixed structure $\mathcal{S}$.

Lemma 5.9. *Let Q be an RC query with pairwise distinct (free and bound) variables and let $\mathsf{rw}(Q) = (\hat{Q}_{fin}, \hat{Q}_{inf})$. Let $\hat{Q} \in \{\hat{Q}_{fin}, \hat{Q}_{inf}\}$. Then*

$$\mathsf{cost}^{\mathcal{S}}(\hat{Q}) \leq \left|\mathsf{sub}(\hat{Q})\right| \cdot \left|\mathsf{av}(\hat{Q})\right| \cdot 2^{|\mathsf{av}(\hat{Q})|} \cdot \textstyle\sum_{\mathcal{Q}_{qps} \subseteq \overline{\mathsf{qps}}(Q), \mathsf{minimal}(\mathcal{Q}_{qps})} \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]| \,.$$

Proof. Recall that $|\mathsf{sub}(\hat{Q})|$ denotes the number of subqueries of the query $\hat{Q}$ and thus bounds the number of RANF subqueries $\hat{Q}'$ of the query $\hat{Q}$. For every subquery $\hat{Q}'$ of $\hat{Q}$, we first use the fact that $|\mathsf{fv}(\hat{Q}')| \leq |\mathsf{av}(\hat{Q})|$ to bound $|[\![\hat{Q}']\!]| \cdot |\mathsf{fv}(\hat{Q}')| \leq |[\![\hat{Q}']\!]| \cdot |\mathsf{av}(\hat{Q})|$. Then we use the estimation of $|[\![\hat{Q}']\!]|$ by Lemma 5.8. □

Finally, we prove Theorem 5.2.

Proof of Theorem 5.2. We derive Theorem 5.2 from Lemma 5.9 and the fact that the quantities $\left|\mathsf{sub}(\hat{Q})\right|$, $\left|\mathsf{av}(\hat{Q})\right|$, and $2^{\left|\mathsf{av}(\hat{Q})\right|^2}$ only depend on the query Q and thus they do not contribute to the asymptotic time complexity of capturing a fixed query Q. □

6. Implementation

We have implemented our translation RC2SQL consisting of roughly 1000 lines of OCaml code [RBKT22a]. It consists of multiple translation steps that take an arbitrary relational calculus (RC) query and produce two SQL queries.

Figure 9 summarizes the order of the translation steps and the functions that implement them. The function $\mathsf{split}(\cdot)$ (Section 4.3), applied in the first step, is the main part of our translation. Recall that it takes an arbitrary RC query and returns two safe-range RC queries. Next, the function $\mathsf{srnf}(\cdot)$ (Section 6.1) converts both queries to safe-range normal form (SRNF), followed by the function $\mathsf{sr2ranf}(\cdot,\cdot)$ (Section 6.2) that converts SRNF queries into relation algebra normal form (RANF). Both normal forms were defined in Section 3. For simplicity, we define a function $\mathsf{sr2ranf}(\cdot)$ that combines the previous two functions and can be applied to any safe-range RC query. In addition to the worst-case complexity, we further improve our translation's average-case complexity by implementing the optimizations inspired by Claußen et al. [CKMP97]. The function $\mathsf{optcnt}(\cdot)$ (Section 6.3) implements these optimizations on the RANF queries. Finally, to derive SQL queries from the RANF queries we first obtain equivalent relational algebra (RA) expressions following a (slightly modified) standard approach [AHV95] implemented by the function $\mathsf{ranf2ra}(\cdot)$ (Section 6.4). To translate the RA expressions into SQL, we reuse a publicly available RA interpreter radb [Yan19] (Section 6.5). We name the composition of the last two steps $\mathsf{ranf2sql}(\cdot)$.

To resolve the nondeterministic choices present in our algorithms (Section 6.6) we always choose the alternative with the lowest query cost. The query cost is estimated by using a sample structure of constant size, called a *training database*. 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 independently of the choice of the training database.

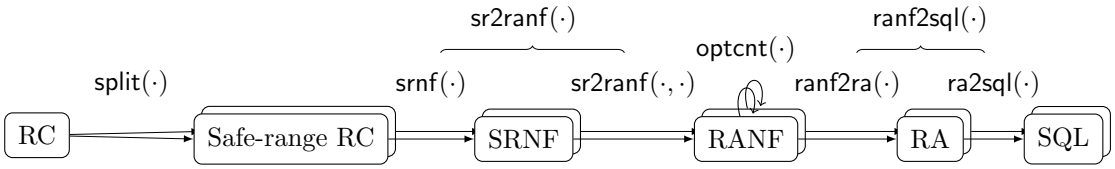

Figure 9: Overview of the functions used in our implementation.

input: An RC query Q.
output: A SRNF query Q_{srnf} such that $Q \equiv Q_{srnf}$, $\mathsf{fv}(Q) = \mathsf{fv}(Q_{srnf})$.

```
1  function srnf(Q) =
2  | switch Q do
3  | | case ¬Q′ do
4  | | | switch Q′ do
5  | | | | case ¬Q″ do return srnf(Q″);
6  | | | | case Q1 ∨ Q2 do return srnf((¬Q1) ∧ (¬Q2));
7  | | | | case Q1 ∧ Q2 do return srnf((¬Q1) ∨ (¬Q2));
8  | | | | case ∃v⃗. Qv⃗ do
9  | | | | | if v⃗ ∩ fv(Qv⃗) = ∅ then return srnf(¬Qv⃗);
10 | | | | | else
11 | | | | | | switch srnf(Qv⃗) do
12 | | | | | | | case Q1 ∨ Q2 do return srnf((¬∃v⃗. Q1) ∧ (¬∃v⃗. Q2));
13 | | | | | | | otherwise do return ¬∃v⃗ ∩ fv(Qv⃗). srnf(Qv⃗);
14 | | | | otherwise do return ¬srnf(Q′);
15 | | case Q1 ∨ Q2 do return srnf(Q1) ∨ srnf(Q2);
16 | | case Q1 ∧ Q2 do return srnf(Q1) ∧ srnf(Q2);
17 | | case ∃v⃗. Qv⃗ do
18 | | | switch srnf(Qv⃗) do
19 | | | | case Q1 ∨ Q2 do return srnf((∃v⃗. Q1) ∨ (∃v⃗. Q2));
20 | | | | otherwise do return ∃v⃗ ∩ fv(Qv⃗). srnf(Qv⃗);
21 | | otherwise do return Q;
```

Figure 10: Translation to SRNF.

Overall, the translation is formally defined as

$$\mathsf{RC2SQL}(Q) \coloneqq (Q'_{fin}, Q'_{inf})$$

where $Q'_{fin} \coloneqq \mathsf{ranf2sql}(\mathsf{optcnt}(\mathsf{sr2ranf}(Q_{fin})))$, $Q'_{inf} \coloneqq \mathsf{ranf2sql}(\mathsf{optcnt}(\mathsf{sr2ranf}(Q_{inf})))$, and $(Q_{fin}, Q_{inf}) \coloneqq \mathsf{split}(Q)$.

6.1. **Translation to SRNF.** Figure 10 defines the function $\mathsf{srnf}(Q)$ that yields a SRNF query equivalent to Q. The function $\mathsf{srnf}(Q)$ pushes negations downwards (Lines 6–7), eliminates double negations (Line 5), drops bound variables that do not occur in the query (Line 9), and distributes existential quantifiers over disjunction (Line 19). The termination of $\mathsf{srnf}(Q)$ follows using the measure $\mathsf{m}(Q)$, shown in Figure 11, that decreases for proper subqueries, after pushing negations and distributing existential quantification over disjunction.

Next we prove a lemma that we use as a precondition for translating safe-range queries in SRNF to queries in RANF.

$$
\begin{array}{ll}
\mathsf{m}(\bot) = \mathsf{m}(\top) & = \mathsf{m}(x \approx t) = 1 \\
\mathsf{m}(r(t_1, \ldots, t_{\iota(r)})) & = 1 \\
\mathsf{m}(\neg Q) & = 2 \cdot \mathsf{m}(Q) \\
\mathsf{m}(Q_1 \vee Q_2) & = 2 \cdot \mathsf{m}(Q_1) + 2 \cdot \mathsf{m}(Q_2) + 2 \\
\mathsf{m}(Q_1 \wedge Q_2) & = \mathsf{m}(Q_1) + \mathsf{m}(Q_2) + 1 \\
\mathsf{m}(\exists x.\, Q_x) & = 2 \cdot \mathsf{m}(Q_x)
\end{array}
$$

Figure 11: Measure on RC queries.

Lemma 6.1. *Let Q_{srnf} be a query in SRNF. Then $\mathsf{gen}(x, \neg Q')$ does not hold for any variable x and subquery $\neg Q'$ of Q_{srnf}.*

Proof. Using Figure 4, $\mathsf{gen}(x, \neg Q')$ can only hold if $\neg Q'$ has the form $\neg\neg Q$, $\neg(Q_1 \vee Q_2)$, or $\neg(Q_1 \wedge Q_2)$. The SRNF query Q_{srnf} cannot have a subquery $\neg Q'$ that has any such form. □

6.2. **Translation to RANF.** The function $\mathsf{sr2ranf}(Q, \mathcal{Q}) = (\hat{Q}, \overline{\mathcal{Q}})$, defined in Figure 12, where $\mathsf{sr2ranf}$ stands for *safe range to relational algebra normal form*, takes a safe-range query $Q \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ in SRNF, or in existential normal form (ENF) (see Appendix B) and returns a RANF query $\hat{Q}$ such that $Q \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q} \equiv \hat{Q} \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$. To restrict variables in Q, the function $\mathsf{sr2ranf}(Q, \mathcal{Q})$ conjoins a subset of queries $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ to Q. Given a safe-range query Q, we first convert Q into SRNF and set $\mathcal{Q} = \varnothing$. Then we define $\mathsf{sr2ranf}(Q) \coloneqq \hat{Q}$, where $(\hat{Q}, \_) \coloneqq \mathsf{sr2ranf}(\mathsf{srnf}(Q), \varnothing)$, to be a RANF query $\hat{Q}$ equivalent to Q. The termination of $\mathsf{sr2ranf}(Q, \mathcal{Q})$ follows from the lexicographic measure $(2 \cdot \mathsf{m}(Q) + \mathsf{eqneg}(Q) + 2 \cdot \sum_{\overline{Q} \in \mathcal{Q}} \mathsf{m}(\overline{Q}) + 2 \cdot |\mathcal{Q}|, \mathsf{m}(Q) + \sum_{\overline{Q} \in \mathcal{Q}} \mathsf{m}(\overline{Q}))$. Here $\mathsf{m}(Q)$ is defined in Figure 11, $\mathsf{eqneg}(Q) \coloneqq 1$ if Q is an equality between two variables or the negation of a query, and $\mathsf{eqneg}(Q) \coloneqq 0$ otherwise.

Next we describe the definition of $\mathsf{sr2ranf}(Q, \mathcal{Q})$ that follows [AHV95, Algorithm 5.4.7]. Note that no constant propagation (Figure 3) is needed in [AHV95, Algorithm 5.4.7], because the constants $\bot$ and $\top$ are not in the query syntax [AHV95, Section 5.3]. Because $\mathsf{gen}(x, \bot)$ holds and $x \notin \mathsf{fv}(\bot)$, we need to perform constant propagation to guarantee that every disjunct has the same set of free variables (e.g., the query $\bot \vee \mathsf{B}(x)$ must be translated to $\mathsf{B}(x)$ to be in RANF). We flatten the disjunction and conjunction using $\mathsf{flat}^\vee(\cdot)$ and $\mathsf{flat}^\wedge(\cdot)$, respectively. In the case of a conjunction $Q^\wedge$, we first split the queries from $\mathsf{flat}^\wedge(Q^\wedge)$ and $\mathcal{Q}$ into queries $\mathcal{Q}^+$ that do not have the form of a negation and queries $\mathcal{Q}^-$ that do. Then we take out equalities between two variables and negations of equalities between two variables from the sets $\mathcal{Q}^+$ and $\mathcal{Q}^-$, respectively. To partition $\mathsf{flat}^\wedge(Q^\wedge) \cup \mathcal{Q}$ this way, we define the predicates $\mathsf{neg}(Q)$ and $\mathsf{eq}(Q)$ characterizing equalities between two variables and negations, respectively, i.e., $\mathsf{neg}(Q)$ is true iff Q has the form $\neg Q'$ and $\mathsf{eq}(Q)$ is true iff Q has the form $x \approx y$. Finally, the function $\mathsf{sort}^\wedge(\mathcal{Q})$ converts a set of queries into a RANF conjunction, defined in Figure 6, i.e., a left-associative conjunction in RANF. Note that the function $\mathsf{sort}^\wedge(\mathcal{Q})$ must order the queries $x \approx y$ so that either x or y is free in some preceding conjunct, e.g., $\mathsf{B}(x) \wedge x \approx y \wedge y \approx z$ is in RANF, but $\mathsf{B}(x) \wedge y \approx z \wedge x \approx y$ is not. In the case of an existentially quantified query $\exists \vec{v}.\, Q_{\vec{v}}$, we rename the variables $\vec{v}$ to avoid a clash of the free variables in the set of queries $\mathcal{Q}$ with the bound variables $\vec{v}$.

Finally, we resolve the nondeterministic choices in $\mathsf{sr2ranf}(Q, \mathcal{Q})$ by minimizing the cost of the resulting RANF query with respect to a training database (Section 6.6).

input: A safe-range query $Q \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ such that for all subqueries of the form $\neg Q'$, $\mathsf{gen}(x, \neg Q')$ does not hold for any variable x.

output: A RANF query $\hat{Q}$ and a subset of queries $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ such that $Q \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q} \equiv \hat{Q} \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$; for all $\mathcal{S}$ and α, $(\mathcal{S}, \alpha) \models \hat{Q} \implies (\mathcal{S}, \alpha) \models \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ holds; $\hat{Q} = \mathsf{cp}(\hat{Q})$; and $\mathsf{fv}(Q) \subseteq \mathsf{fv}(\hat{Q}) \subseteq \mathsf{fv}(Q) \cup \mathsf{fv}(\mathcal{Q})$, unless $\hat{Q} = \bot$.

function $\mathsf{sr2ranf}(Q, \mathcal{Q}) =$
  if $\mathsf{ranf}(Q)$ **then**
    return $(\mathsf{cp}(Q), \varnothing)$
  switch Q **do**
    case $x \approx y$ **do**
      return $\mathsf{sr2ranf}(x \approx y \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}, \varnothing)$
    case $\neg Q'$ **do**
      $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q} \mid (\neg Q') \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe range$\}$;
      if $\overline{\mathcal{Q}} = \varnothing$ **then**
        $(\hat{Q}', \_) := \mathsf{sr2ranf}(Q', \varnothing)$;
        return $(\mathsf{cp}(\neg \hat{Q}'), \varnothing)$;
      else return $\mathsf{sr2ranf}((\neg Q') \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}, \varnothing)$;
    case $Q_1 \vee Q_2$ **do**
      $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q} \mid \bigvee_{Q' \in \mathsf{flat}^\vee(Q)} (Q' \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q})$ is safe range$\}$;
      foreach $Q' \in \mathsf{flat}^\vee(Q)$ **do** $(\hat{Q}', \_) := \mathsf{sr2ranf}(Q' \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}, \varnothing)$;
      return $(\mathsf{cp}(\bigvee_{Q' \in \mathsf{flat}^\vee(Q)} \hat{Q}'), \overline{\mathcal{Q}})$;
    case $Q_1 \wedge Q_2$ **do**
      $\mathcal{Q}^- := \{Q' \in \mathsf{flat}^\wedge(Q) \cup \mathcal{Q} \mid \mathsf{neg}(Q')\}$; $\mathcal{Q}^+ := (\mathsf{flat}^\wedge(Q) \cup \mathcal{Q}) \setminus \mathcal{Q}^-$;
      $\mathcal{Q}^{\approx} := \{Q' \in \mathcal{Q}^+ \mid \mathsf{eq}(Q')\}$; $\mathcal{Q}^+ := \mathcal{Q}^+ \setminus \mathcal{Q}^{\approx}$;
      $\mathcal{Q}^{\not\approx} := \{\neg Q' \in \mathcal{Q}^- \mid \mathsf{eq}(Q')\}$; $\mathcal{Q}^- := \mathcal{Q}^- \setminus \mathcal{Q}^{\not\approx}$;
      foreach $Q' \in \mathcal{Q}^+$ **do** $(\hat{Q}', \mathcal{Q}_{Q'}) := \mathsf{sr2ranf}(Q', (\mathcal{Q}^+ \cup \mathcal{Q}^{\approx}) \setminus \{Q'\})$;
      foreach $\neg Q' \in \mathcal{Q}^-$ **do** $(\hat{Q}', \_) := \mathsf{sr2ranf}(Q', \mathcal{Q}^+ \cup \mathcal{Q}^{\approx})$;
      $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q}^+ \mid \mathcal{Q}^+ \subseteq \bigcup_{Q' \in \overline{\mathcal{Q}}} (\mathcal{Q}_{Q'} \cup \{Q'\})\}$;
      return $(\mathsf{cp}(\mathsf{sort}^\wedge(\bigcup_{Q' \in \overline{\mathcal{Q}}} \{\hat{Q}'\} \cup \mathcal{Q}^{\approx} \cup \bigcup_{\neg Q' \in \mathcal{Q}^-} \{\neg \hat{Q}'\} \cup \mathcal{Q}^{\not\approx})), \bigcup_{Q' \in \overline{\mathcal{Q}}} (\mathcal{Q}_{Q'} \cap \mathcal{Q}))$;
    case $\exists \vec{v}.\, Q_{\vec{v}}$ **do**
      if $\mathsf{fv}(\mathcal{Q}) \cap \vec{v} \neq \varnothing$ **then** $\vec{w} \leftarrow \{\vec{w} \mid |\vec{w}| = |\vec{v}|$ and $((\mathsf{fv}(Q_{\vec{v}}) \setminus \vec{v}) \cup \mathsf{fv}(\mathcal{Q})) \cap \vec{w} = \varnothing\}$;
      else $\vec{w} := \vec{v}$;
      $Q_{\vec{w}} := Q_{\vec{v}}[\vec{v} \mapsto \vec{w}]$;
      $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q} \mid Q_{\vec{w}} \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe range$\}$;
      $(\hat{Q}_{\vec{w}}, \_) := \mathsf{sr2ranf}(Q_{\vec{w}} \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}, \varnothing)$;
      return $(\mathsf{cp}(\exists \vec{w}.\, \hat{Q}_{\vec{w}}), \overline{\mathcal{Q}})$;
    otherwise do return $(\mathsf{cp}(Q), \varnothing)$;

Figure 12: Translation of a safe-range query in SRNF to RANF.

6.3. **Optimization using Count Aggregations.** In this section, we introduce count aggregations and describe a generalization of Claußen et al. [CKMP97]'s approach to evaluate RANF queries using count aggregations. Consider the query

$$Q_x \wedge \neg\exists y.\, (Q_x \wedge Q_y \wedge \neg Q_{xy}),$$

where $\mathsf{fv}(Q_x) = \{x\}$, $\mathsf{fv}(Q_y) = \{y\}$, and $\mathsf{fv}(Q_{xy}) = \{x, y\}$. This query is obtained by applying our translation to the query $Q_x \wedge \forall y.\, (Q_y \longrightarrow Q_{xy})$. The cost of the translated query is dominated by the cost of the Cartesian product $Q_x \wedge Q_y$. Consider the subquery $Q' \coloneqq \exists y.\, (Q_x \wedge Q_y \wedge \neg Q_{xy})$. A assignment α satisfies Q' iff α satisfies Q_x and there exists a value d such that $\alpha[y \mapsto d]$ satisfies Q_y, but not Q_{xy}, i.e., the number of values d such that $\alpha[y \mapsto d]$ satisfies Q_y is not equal to the number of values d such that $\alpha[y \mapsto d]$ satisfies both Q_y and Q_{xy}. An alternative evaluation of Q' evaluates the queries Q_x, Q_y, $Q_y \wedge Q_{xy}$ and computes the numbers of values d such that $\alpha[y \mapsto d]$ satisfies Q_y and $Q_y \wedge Q_{xy}$, respectively, i.e., computes count aggregations. These count aggregations are then used to filter assignments α satisfying Q_x to get assignments α satisfying Q'. The asymptotic time complexity of the alternative evaluation never exceeds that of the evaluation computing the Cartesian product $Q_x \wedge Q_y$ and asymptotically improves it if $|[\![Q_x]\!]| + |[\![Q_y]\!]| + |[\![Q_{xy}]\!]| \ll |[\![Q_x \wedge Q_y]\!]|$. Furthermore, we observe that a assignment α satisfies $Q_x \wedge \neg Q'$ if α satisfies Q_x, but not Q', i.e., the number of values d such that $\alpha[y \mapsto d]$ satisfies Q_y is equal to the number of values d such that $\alpha[y \mapsto d]$ satisfies $Q_y \wedge Q_{xy}$.

Next we introduce the syntax and semantics of count aggregations. We extend RC's syntax by $[\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c)$, where Q is a query, c is a variable representing the result of the count aggregation, and $\vec{v}$ is a sequence of variables that are bound by the aggregation operator. The semantics of the count aggregation is defined as follows:

$$(\mathcal{S}, \alpha) \models [\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c) \;\text{ iff }\; (M = \varnothing \longrightarrow \mathsf{fv}(Q) \subseteq \vec{v}) \,\text{and}\, \alpha(c) = |M|\,,$$

where $M = \{\vec{d} \in \mathcal{D}^{|\vec{v}|} \mid (\mathcal{S}, \alpha[\vec{v} \mapsto \vec{d}]) \models Q\}$. We use the condition $M = \varnothing \longrightarrow \mathsf{fv}(Q) \subseteq \vec{v}$ instead of $M \neq \varnothing$ to set c to a zero count if the group M is empty and there are no group-by variables (like in SQL). The set of free variables in a count aggregation is $\mathsf{fv}([\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c)) = (\mathsf{fv}(Q) \setminus \vec{v}) \cup \{c\}$. Finally, we extend the definition of $\mathsf{ranf}(Q)$ with the case of a count aggregation:

$$\mathsf{ranf}([\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c)) \;\text{ iff }\; \mathsf{ranf}(Q) \,\text{and}\, \vec{v} \subseteq \mathsf{fv}(Q) \,\text{and}\, c \notin \mathsf{fv}(Q).$$

We formulate translations introducing count aggregations in the following two lemmas.

Lemma 6.2. *Given $\mathcal{Q} \neq \varnothing$, let $\exists \vec{v}.\, Q_{\vec{v}} \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \neg \overline{Q}$ be a RANF query. Let c, c' be fresh variables that do not occur in $\mathsf{fv}(Q_{\vec{v}})$. Then*

$$\begin{aligned}(\exists \vec{v}.\, Q_{\vec{v}} \wedge \textstyle\bigwedge_{\overline{Q} \in \mathcal{Q}} \neg \overline{Q}) \equiv\; & ((\exists \vec{v}.\, Q_{\vec{v}}) \wedge \textstyle\bigwedge_{\overline{Q} \in \mathcal{Q}} \neg(\exists \vec{v}.\, Q_{\vec{v}} \wedge \overline{Q})) \vee \\ & (\exists c, c'.\, [\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c) \wedge \\ & \qquad [\mathsf{CNT}\, \vec{v}.\, \textstyle\bigvee_{\overline{Q} \in \mathcal{Q}} (Q_{\vec{v}} \wedge \overline{Q})](c') \wedge \neg(c = c')).\end{aligned} \tag{\#}$$

Moreover, the right-hand side of (#) is in RANF.

Lemma 6.3. *Given $\mathcal{Q} \neq \varnothing$, let $\hat{Q} \wedge \neg\exists\vec{v}.\, Q_{\vec{v}} \wedge \bigwedge_{\overline{Q}\in\mathcal{Q}} \neg\overline{Q}$, be a RANF query. Let c, c' be fresh variables that do not occur in $\mathsf{fv}(\hat{Q}) \cup \mathsf{fv}(Q_{\vec{v}})$. Then*

$$\begin{aligned}(\hat{Q} \wedge \neg\exists\vec{v}.\, Q_{\vec{v}} \wedge \textstyle\bigwedge_{\overline{Q}\in\mathcal{Q}} \neg\overline{Q}) \equiv\ & (\hat{Q} \wedge \neg(\exists\vec{v}.\, Q_{\vec{v}})) \vee \\ & (\exists c, c'.\, \hat{Q} \wedge [\mathsf{CNT}\,\vec{v}.\, Q_{\vec{v}}](c) \wedge \\ & \qquad [\mathsf{CNT}\,\vec{v}.\, \textstyle\bigvee_{\overline{Q}\in\mathcal{Q}}(Q_{\vec{v}} \wedge \overline{Q})](c') \wedge (c = c')).\end{aligned} \tag{\#\#}$$

Moreover, the right-hand side of (##) is in RANF.

Note that the query cost does not decrease after applying the translation (#) or (##) because of the subquery $[\mathsf{CNT}\,\vec{v}.\, Q_{\vec{v}}](c)$ in which $Q_{\vec{v}}$ is evaluated before the count aggregation is computed. For the query $\exists y.\, ((Q_x \wedge Q_y) \wedge \neg Q_{xy})$ from before, we would compute $[\mathsf{CNT}\, y.\, Q_x \wedge Q_y](c)$, i.e., we would not (yet) avoid computing the Cartesian product $Q_x \wedge Q_y$. However, we could reduce the scope of the bound variable y by further translating

$$[\mathsf{CNT}\, y.\, Q_x \wedge Q_y](c) \equiv Q_x \wedge [\mathsf{CNT}\, y.\, Q_y](c).$$

This technique, called *mini-scoping*, can be applied to a count aggregation $[\mathsf{CNT}\,\vec{v}.\, Q_{\vec{v}}](c)$ if the aggregated query $Q_{\vec{v}}$ is a conjunction that can be split into two RANF conjuncts and the variables $\vec{v}$ do not occur free in one of the conjuncts (that conjunct can be pulled out of the count aggregation). Mini-scoping can be analogously applied to queries of the form $\exists\vec{v}.\, Q_{\vec{v}}$.

Moreover, we can split a count aggregation over a conjunction $Q_{\vec{v}} \wedge Q'_{\vec{v}}$ into a product of count aggregations if the conjunction can be split into two RANF conjuncts with disjoint sets of bound variables, i.e., $\vec{v} \cap \mathsf{fv}(Q_{\vec{v}}) \cap \mathsf{fv}(Q'_{\vec{v}}) = \varnothing$:

$$[\mathsf{CNT}\,\vec{v}.\, Q_{\vec{v}} \wedge Q'_{\vec{v}}](c) \equiv (\exists c_1, c_2.\, [\mathsf{CNT}\,\vec{v} \cap \mathsf{fv}(Q_{\vec{v}}).\, Q_{\vec{v}}](c_1) \wedge [\mathsf{CNT}\,\vec{v} \cap \mathsf{fv}(Q'_{\vec{v}}).\, Q'_{\vec{v}}](c_2) \wedge c = c_1 \cdot c_2).$$

Here c_1 and c_2 are fresh variables that do not occur in $\mathsf{fv}(Q_{\vec{v}}) \cup \mathsf{fv}(Q'_{\vec{v}}) \cup \{c\}$. Note that mini-scoping is only a heuristic and it can both improve and harm the time complexity of query evaluation. We leave the application of other more general optimization algorithms [KNR16, OS16]) as future work.

We implement the translations from Lemmas 6.2 and 6.3 and mini-scoping in the function $\mathsf{optcnt}(\cdot)$. Given a RANF query $\hat{Q}$, $\mathsf{optcnt}(\hat{Q})$ is an equivalent RANF query after introducing count aggregations and performing mini-scoping. The function $\mathsf{optcnt}(\hat{Q})$ uses a training database to decide how to apply the translations from Lemmas 6.2 and 6.3 and mini-scoping. More specifically, the function $\mathsf{optcnt}(\hat{Q})$ tries several possibilities and chooses one that minimizes the query cost of the resulting RANF query.

Example 6.4. *We show how to introduce count aggregations into the RANF query*

$$\hat{Q} \coloneqq Q_x \wedge \neg\exists y.\, (Q_x \wedge Q_y \wedge \neg Q_{xy}).$$

After applying the translation (##) and mini-scoping to this query, we obtain the following equivalent RANF query:

$$\begin{aligned}\mathsf{optcnt}(\hat{Q}) \coloneqq\ & (Q_x \wedge \neg(Q_x \wedge \exists y.\, Q_y)) \vee \\ & (\exists c, c'.\, Q_x \wedge\ [\mathsf{CNT}\, y.\, Q_y](c) \wedge [\mathsf{CNT}\, y.\, Q_y \wedge Q_{xy}](c') \wedge (c = c')).\end{aligned}$$

6.4. **Translating RANF to RA.** Our translation of a RANF query into SQL has two steps: we first translate the query to an equivalent RA expression, which we then translate to SQL using a publicly available RA interpreter radb [Yan19].

We define the function $\mathsf{ranf2ra}(\hat{Q})$ translating RANF queries $\hat{Q}$ into equivalent RA expressions $\mathsf{ranf2ra}(\hat{Q})$. The translation is based on Algorithm 5.4.8 by Abiteboul et al. [AHV95], which we modify as follows. We adjust the way closed RC queries are handled. Chomicki and Toman [CT95] observed that closed RC queries cannot be handled by SQL, since SQL allows neither empty projections nor 0-ary relations. They propose to use a unary auxiliary predicate $\mathsf{A} \in \mathcal{R}$ whose interpretation $\mathsf{A}^{\mathcal{S}} = \{\mathsf{t}\}$ always contains exactly one tuple t. Every closed query $\exists x.\, Q_x$ is then translated into $\exists x.\, \mathsf{A}(t) \wedge Q_x$ with an auxiliary free variable t. Every other closed query $\hat{Q}$ is translated into $\mathsf{A}(t) \wedge \hat{Q}$, e.g., $\mathsf{B}(42)$ is translated into $\mathsf{A}(t) \wedge \mathsf{B}(42)$. We also use the auxiliary predicate A to translate queries of the form $x \approx \mathsf{c}$ and $\mathsf{c} \approx x$ because the single tuple (t) in $\mathsf{A}^{\mathcal{S}}$ can be mapped to any constant c. Finally, we extend [AHV95, Algorithm 5.4.8] with queries of the form $[\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c)$.

6.5. **Translating RA to SQL.** The radb interpreter, abbreviated here by the function $\mathsf{ra2sql}(\cdot)$, translates an RA expression into SQL by simply mapping the RA connectives into their SQL counterparts. The function $\mathsf{ra2sql}(\cdot)$ is primitive recursive on RA expressions. We modify radb to further improve performance of the query evaluation as follows.

A RANF query $Q_1 \wedge \neg Q_2$, where $\mathsf{ranf}(Q_1)$, $\mathsf{ranf}(Q_2)$, and $\mathsf{fv}(Q_2) \subseteq \mathsf{fv}(Q_1)$ is translated into RA expression $\mathsf{ranf2ra}(Q_1) \triangleright \mathsf{ranf2ra}(Q_2)$, where $\triangleright$ denotes the anti-join operator and $\mathsf{ranf2ra}(Q_1)$, $\mathsf{ranf2ra}(Q_2)$ are the equivalent relational algebra expressions for Q_1, Q_2, respectively. The radb interpreter only supports the anti-join operator $\mathsf{ranf2ra}(Q_1) \triangleright \mathsf{ranf2ra}(Q_2)$ expressed as $\mathsf{ranf2ra}(Q_1) - (\mathsf{ranf2ra}(Q_1) \bowtie \mathsf{ranf2ra}(Q_2))$, where $-$ denotes the set difference operator and $\bowtie$ denotes the natural join. Alternatively, the anti-join operator can be directly mapped to `LEFT JOIN` in SQL. We generalize radb to use `LEFT JOIN` since it performs better in our empirical evaluation [RBKT22a].

The radb interpreter introduces a separate SQL subquery in a `WITH` clause for every subexpression in the RA expression. We extend radb to additionally perform common subquery elimination, i.e., to merge syntactically equal subqueries. Common subquery elimination is also assumed in our query cost (Section 3.5).

Finally, the function $\mathsf{ranf2sql}(\hat{Q})$ is defined as $\mathsf{ranf2sql}(\hat{Q}) \coloneqq \mathsf{ra2sql}(\mathsf{ranf2ra}(\hat{Q}))$, i.e., as a composition of the two translations from RANF to RA and from RA to SQL.

6.6. **Resolving Nondeterministic Choices.** To resolve the nondeterministic choices in our algorithms, we suppose that the algorithms have access to a *training database* $\mathcal{T}$ of constant size. The training database is used to compare the cost of queries over the actual database and thus it should preserve the relative ordering of queries by their cost over the actual database as much as possible. Still, our translation satisfies the correctness and worst-case complexity claims (Section 4.3 and 5) for every choice of the training database. The training databases used in our empirical evaluation are obtained using the function dg (Section 7) with $|\mathcal{T}^+| = |\mathcal{T}^-| = 2$. Because of its constant size, the complexity of evaluating a query over the training database is constant and does not impact the asymptotic time complexity of evaluating the query over the actual database using our translation. There are two types of nondeterministic choices in our algorithms:

input: An RC query Q satisfying CON, CST, VAR, REP, $\gamma \in \{0, 1\}$.
output: Two sets of variables $\mathcal{V}^+$ and $\mathcal{V}^-$ whose values must be equal in every tuple in $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$ when computing a Data Golf structure.

function $\mathsf{dg}^{\approx}(Q, \gamma) =$
  switch Q **do**
    case $r(t_1, \ldots, t_{\iota(r)})$ **do return** $(\varnothing, \varnothing)$;
    case $x \approx y$ **do return** $(\{x, y\}, \varnothing)$;
    case $\neg Q'$ **do**
      $(\mathcal{V}^+, \mathcal{V}^-) \coloneqq \mathsf{dg}^{\approx}(Q', \gamma)$;
      return $(\mathcal{V}^-, \mathcal{V}^+)$;
    case $Q'_1 \vee Q'_2$ *or* $Q'_1 \wedge Q'_2$ **do**
      $(\mathcal{V}^+_1, \mathcal{V}^-_1) \coloneqq \mathsf{dg}^{\approx}(Q'_1, \gamma)$;
      $(\mathcal{V}^+_2, \mathcal{V}^-_2) \coloneqq \mathsf{dg}^{\approx}(Q'_2, \gamma)$;
      if $\gamma = 0$ **then return** $(\mathcal{V}^+_1 \cup \mathcal{V}^+_2, \mathcal{V}^-_1 \cup \mathcal{V}^-_2)$;
      else if $Q = Q'_1 \vee Q'_2$ **then return** $(\mathcal{V}^+_1 \cup \mathcal{V}^-_2, \mathcal{V}^-_1 \cup \mathcal{V}^-_2)$;
      else if $Q = Q'_1 \wedge Q'_2$ **then return** $(\mathcal{V}^+_1 \cup \mathcal{V}^+_2, \mathcal{V}^+_1 \cup \mathcal{V}^-_2)$;
    case $\exists y.\, Q_y$ **do return** $\mathsf{dg}^{\approx}(Q_y, \gamma)$;

Figure 13: Computing sets of variables $\mathcal{V}^+$ and $\mathcal{V}^-$ to reflect equalities in a query when computing a Data Golf structure.

- Choosing some $X \leftarrow \mathcal{X}$ in a while-loop. As the while-loops always update $\mathcal{X}$ with $\mathcal{X} \coloneqq (\mathcal{X} \setminus \{X\}) \cup f(X)$ for some f, the order in which the elements of $\mathcal{X}$ are chosen does not matter.
- Choosing a subset of queries $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ in the function $\mathsf{sr2ranf}(Q, \mathcal{Q})$. Because $\mathsf{sr2ranf}(Q, \mathcal{Q})$ yields a RANF query, we enumerate all *minimal* subsets (a subset $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ is minimal if there exists no proper subset $\overline{\mathcal{Q}}' \subsetneq \overline{\mathcal{Q}}$ that could be used instead of $\overline{\mathcal{Q}}$) and choose one that minimizes the query cost of the RANF query.
- Choosing a variable $x \in V$ and a set $\mathcal{G}$ such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$, where $\tilde{Q}$ is a query with range-restricted bound variables and $V \subseteq \mathsf{fv}(\tilde{Q})$. Observe that the measure $\mathsf{m}(Q)$ on queries, defined in Figure 11, decreases for the queries in the premises of the rules for $\mathsf{gen}(x, \tilde{Q}, \mathcal{G})$ and $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$, defined in Figures 4 and 5. Hence, deriving $\mathsf{gen}(x, \tilde{Q}, \mathcal{G})$ and $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ either succeeds or gets stuck after at most $\mathsf{m}(\tilde{Q})$ steps. In particular, we can enumerate all sets $\mathcal{G}$ such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ holds. Because we derive one additional query $\tilde{Q}[x \mapsto y]$ for every $y \in \mathsf{eqs}(x, \mathcal{G})$ and a single query $\tilde{Q} \wedge \mathsf{qps}^{\vee}(\mathcal{G})$, we choose $x \in V$ and $\mathcal{G}$ minimizing $|\mathsf{eqs}(x, \mathcal{G})|$ as the first objective and $\sum_{Q_{qp} \in \mathsf{qps}(\mathcal{G})} \mathsf{cost}^{\mathcal{T}}(Q_{qp})$ as the second objective. Our particular choice of $\mathcal{G}$ with $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ is merely a heuristic and does not provide any additional guarantees compared to every other choice of $\mathcal{G}$ with $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$.

This process can be further improved by adopting query plan search heuristics as is done by most of the existing database management systems.

7. Data Golf Benchmark

In this section, we describe the *Data Golf* benchmark, which we use to generate structures (i.e., database instances) for our empirical evaluation. The technical description of

input: An RC query Q satisfying CON, CST, VAR, REP, a sequence of pairwise distinct variables $\vec{v}$, $\mathsf{av}(Q) \subseteq \vec{v}$, sets of tuples $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$ over $\vec{v}$ such that all values of variables from $\mathsf{av}(Q)$ in these tuples are pairwise distinct (also across tuples) except that, in every tuple in $\mathcal{T}^+_{\vec{v}}$ ($\mathcal{T}^-_{\vec{v}}$), the variables in $\mathcal{V}^+$ ($\mathcal{V}^-$) have the same value (which is different across tuples), where $\mathsf{dg}^{\approx}(Q,\gamma) = (\mathcal{V}^+, \mathcal{V}^-)$, $\gamma \in \{0,1\}$.

output: A structure $\mathcal{S}$ such that $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq [\![Q]\!]$, $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \varnothing$, and $[\![Q']\!]$ and $[\![\neg Q']\!]$ contain at least $\min\{|\mathcal{T}^+_{\vec{v}}|, |\mathcal{T}^-_{\vec{v}}|\}$ 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 $r(t_1, \ldots, t_{\iota(r)})$ **do return** $\{r^{\mathcal{S}} \mapsto \mathcal{T}^+_{\vec{v}}[t_1, \ldots, t_{\iota(r)}]\}$;
    case $x \approx y$ **do return** $\varnothing$;
    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**
      $(\mathcal{V}^+_1, \mathcal{V}^-_1) \coloneqq \mathsf{dg}^{\approx}(Q'_1, \gamma)$; $(\mathcal{V}^+_2, \mathcal{V}^-_2) \coloneqq \mathsf{dg}^{\approx}(Q'_2, \gamma)$;
      if $\gamma = 0$ **then** $(\mathcal{V}^1, \mathcal{V}^2) \coloneqq (\mathcal{V}^+_1 \cup \mathcal{V}^-_2, \mathcal{V}^-_1 \cup \mathcal{V}^+_2)$;
      else if $Q = Q'_1 \wedge Q'_2$ **then** $(\mathcal{V}^1, \mathcal{V}^2) \coloneqq (\mathcal{V}^-_1 \cup \mathcal{V}^-_2, \mathcal{V}^-_1 \cup \mathcal{V}^+_2)$;
      else if $Q = Q'_1 \vee Q'_2$ **then** $(\mathcal{V}^1, \mathcal{V}^2) \coloneqq (\mathcal{V}^+_1 \cup \mathcal{V}^+_2, \mathcal{V}^-_1 \cup \mathcal{V}^+_2)$;
      $(\mathcal{T}^1_{\vec{v}}, \mathcal{T}^2_{\vec{v}}) \leftarrow \{(\mathcal{T}^1_{\vec{v}}, \mathcal{T}^2_{\vec{v}}) \mid |\mathcal{T}^1_{\vec{v}}| = |\mathcal{T}^2_{\vec{v}}| = \min\{|\mathcal{T}^+_{\vec{v}}|, |\mathcal{T}^-_{\vec{v}}|\},$ all values in tuples in $\mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \mathcal{T}^1_{\vec{v}}, \mathcal{T}^2_{\vec{v}}$ are chosen to be pairwise distinct (also across tuples) except that, in every tuple in $\mathcal{T}^1_{\vec{v}}$ ($\mathcal{T}^2_{\vec{v}}$), the variables in $\mathcal{V}^1$ ($\mathcal{V}^2$) have the same value (which is different across tuples)$\}$;
      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 if $Q = Q'_1 \vee Q'_2$ **then return** $\mathsf{dg}(Q'_1, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \gamma) \cup \mathsf{dg}(Q'_2, \vec{v}, \mathcal{T}^1_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^+_{\vec{v}}, \gamma)$;
      else if $Q = Q'_1 \wedge Q'_2$ **then 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 return** $\mathsf{dg}(Q_y, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$;

Figure 14: Computing the Data Golf structure.

this benchmark is only needed to fully understand the setup of our empirical evaluation (Section 8), but its details are independent of our query translation (Section 4–6).

Given an RC query, we seek a structure that yields a nontrivial evaluation result for the overall query and for all its subqueries. Intuitively, the structure makes query evaluation potentially more challenging compared to the case where some subquery evaluates to a trivial (e.g., empty) result. More specifically, Data Golf has two objectives. The first resembles the *regex golf* game's objective [Ell13] (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.

Formally, given a query Q and two sets of tuples $\mathcal{T}^+$ and $\mathcal{T}^-$ over a fixed domain $\mathcal{D}$, representing assignments of $\mathsf{av}(Q)$ and satisfying further assumptions on their values, Data Golf

produces a structure $\mathcal{S}$ (represented as a partial mapping from predicate symbols to their interpretations) such that the projections of tuples in $\mathcal{T}^+$ ($\mathcal{T}^-$, respectively) to $\vec{\mathsf{fv}}(Q)$ are in $[\![Q]\!]$ (disjoint from $[\![Q]\!]$, respectively) and $|[\![Q']\!]|$ and $|[\![\neg Q']\!]|$ are at least $\min\{|\mathcal{T}^+|, |\mathcal{T}^-|\}$, for every $Q' \sqsubseteq Q$. To be able to produce such a structure $\mathcal{S}$, we make the following assumptions on Q:

(CON) for every subquery $\exists y.\, Q_y$ of Q we have $\mathsf{con}_{\mathsf{vgt}}(y, Q_y, \mathcal{A})$ (Figure 17) for some set of atomic predicates $\mathcal{A}$ and, moreover, $\{y\} \subsetneq \mathsf{fv}(Q_{ap})$ holds for every $Q_{ap} \in \mathcal{A}$; these conditions prevent subqueries like $\exists y.\, \neg \mathsf{P}_2(x, y)$ and $\exists y.\, (\mathsf{P}_2(x, y) \vee \mathsf{P}_1(y))$, respectively;

(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 and no equalities $x \approx y$ in Q share a variable; this avoids subqueries like $\mathsf{P}_1(x) \wedge \neg \mathsf{P}_1(x)$ and $x \approx y \wedge \neg x \approx y$.

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 $t_1, \ldots, t_k \in \vec{v} \cup \mathcal{C}$ of terms, we denote by $\vec{d}(\vec{v})[t_1, \ldots, t_k]$ the tuple obtained by evaluating the terms $t_1, \ldots, t_k$ over $\vec{d}(\vec{v})$. Formally, we define $\vec{d}(\vec{v})[t_1, \ldots, t_k] \coloneqq (d'_i)_{i=1}^k$, 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 Figure 14, where $\vec{v}$ is a sequence of pairwise distinct variables containing all variables in Q, i.e., $\mathsf{av}(Q) \subseteq \vec{v}$, $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$ are sets of tuples over $\vec{v}$, and $\gamma \in \{0, 1\}$ is a *strategy*. To reflect Q's equalities in the sets $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$, given a strategy γ, we define the function $\mathsf{dg}^{\approx}(Q, \gamma) = (\mathcal{V}^+, \mathcal{V}^-)$ (Figure 13) that computes two sets of variables $\mathcal{V}^+$ and $\mathcal{V}^-$ whose values must be equal in every tuple in $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$, respectively. The values of the remaining variables ($\vec{v} \setminus \mathcal{V}^+$ and $\vec{v} \setminus \mathcal{V}^-$, respectively) must be pairwise distinct and also different from the value of the variables in $\mathcal{V}^+$ and $\mathcal{V}^-$, respectively. 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 $\mathcal{T}^1_{\vec{v}}$, $\mathcal{T}^2_{\vec{v}}$.

Let $\mathcal{S}$ be a Data Golf structure computed by $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$. We justify why $\mathcal{S}$ satisfies $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq [\![Q]\!]$ and $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \varnothing$. 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 $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \varnothing$ 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 $\vec{d} \in [\![Q]\!]$. Because $\vec{d} \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)]$, there exists some d such that $\vec{d} \cdot d \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$. Because $\vec{d} \in [\![Q]\!]$, there exists some d' such that $\vec{d} \cdot d' \in [\![Q_y]\!]$. By the induction hypothesis, $\vec{d} \cdot d \notin [\![Q_y]\!]$ and $\vec{d} \cdot d' \notin \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$. Because $\mathsf{con}_{\mathsf{vgt}}(y, Q_y, \mathcal{A})$ holds for some $\mathcal{A}$ satisfying (CON), the query Q_y is equivalent to $(Q_y \wedge \bigvee_{Q_{ap} \in \mathcal{A}} Q_{ap}) \vee Q_y[y/\bot]$. We have $\vec{d} \cdot d' \in [\![Q_y]\!]$. If the tuple $\vec{d} \cdot d'$ satisfies $Q_y[y/\bot]$, then $\vec{d} \cdot d \in [\![Q_y]\!]$ (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 atomic predicate $Q_{ap} \in \mathcal{A}$ and (CON) implies $\{y\} \subsetneq \mathsf{fv}(Q_{ap})$. Hence, the tuples $\vec{d} \cdot d$ and $\vec{d} \cdot d'$ agree on the assignment of a variable $x \in \mathsf{fv}(Q_{ap}) \setminus \{y\}$. Let $\overline{\mathcal{T}}^+_{\vec{v}}$ and $\overline{\mathcal{T}}^-_{\vec{v}}$ be the sets in the recursive call of dg on the atomic predicate from Q_{ap}. Because $\vec{d} \cdot d \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$ and $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)] \subseteq \overline{\mathcal{T}}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)] \cup \overline{\mathcal{T}}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$,

the tuple $\vec{d} \cdot d$ is in $\overline{\mathcal{T}}^{+}_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)] \cup \overline{\mathcal{T}}^{-}_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$. Because $\vec{d} \cdot d'$ satisfies the quantified predicate Q_{ap}, the tuple $\vec{d} \cdot d'$ is in $\overline{\mathcal{T}}^{+}_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$. Next we observe that the assignments of every variable (in particular, x) in the tuples from the sets $\overline{\mathcal{T}}^{+}_{\vec{v}}$, $\overline{\mathcal{T}}^{-}_{\vec{v}}$ are pairwise distinct (there can only be equal values of variables within a single tuple). 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), and (REP) imply that Q has no closed subquery. Hence, $\mathcal{T}^{+}_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq [\![Q]\!]$ and $\mathcal{T}^{-}_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \varnothing$ imply that $|[\![Q']\!]|$ and $|[\![\neg Q']\!]|$ contain at least $\min\{\left|\mathcal{T}^{+}_{\vec{v}}\right|, \left|\mathcal{T}^{-}_{\vec{v}}\right|\}$ tuples, for every $Q' \sqsubseteq Q$.

Example 7.1. *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* (CON), (CST), (VAR), *and* (REP). *In particular,* $\mathsf{con}_{\mathsf{vgt}}(y, \mathsf{P}_2(x, y) \wedge \neg\mathsf{P}_3(x, y, z), \mathcal{A})$ *holds for* $\mathcal{A} = \{\mathsf{P}_2(x, y)\}$ *with* $\{y\} \subsetneq \mathsf{fv}(\mathsf{P}_2(x, y))$. *We choose* $\vec{v} = (x, z, y)$, $\mathcal{T}^{+}_{\vec{v}} = \{(0, 4, 8), (2, 6, 10)\}$, *and* $\mathcal{T}^{-}_{\vec{v}} = \{(12, 16, 20), (14, 18, 22)\}$. *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.*

For conjunction (a binary operator), two additional sets of tuples are computed: $\mathcal{T}^{1}_{\vec{v}} = \{(24, 28, 32), (26, 30, 34)\}$ *and* $\mathcal{T}^{2}_{\vec{v}} = \{(36, 40, 44), (38, 42, 46)\}$. *Depending on the strategy (*$\gamma = 0$ *or* $\gamma = 1$*), one of the following structures is computed:* $\mathcal{S}_0 = \{\mathsf{P}_2 \mapsto \{(12, 20), (14, 22), (24, 32), (26, 34)\}, \mathsf{P}_3 \mapsto \mathcal{T}^{+}_{xyz}\}$, *or* $\mathcal{S}_1 = \{\mathsf{P}_2 \mapsto \{(12, 20), (14, 22), (0, 8), (2, 10)\}, \mathsf{P}_3 \mapsto \mathcal{T}^{+}_{xyz}\}$, *where* $\mathcal{T}^{+}_{xyz} = \{(0, 8, 4), (2, 10, 6), (24, 32, 28), (26, 34, 30)\}$.

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

8. Empirical Evaluation

We empirically validate the evaluation performance of the queries output by RC2SQL. We also assess RC2SQL's translation time, the average-case time complexity of query evaluation, scalability to large databases, and DBMS interoperability. To this end, we answer the following research questions:

- RQ1 How does RC2SQL's query evaluation perform compared to the state-of-the-art tools on both domain-independent and domain-dependent queries?
- RQ2 How does RC2SQL's query evaluation scale on large synthetic databases?
- RQ3 How does RC2SQL's query evaluation perform on real-world databases?
- RQ4 How does the count aggregation optimization impact RC2SQL's performance?
- RQ5 Can RC2SQL use different DBMSs for query evaluation?
- RQ6 How long does RC2SQL take to translate different queries (without query evaluation)?

We organize our evaluation into five experiments. Four experiments (Small, Medium, Large, and Real) focus on the type and size of the structures we use for query evaluation. The fifth experiment (Infinite) focuses on the evaluation of non-evaluable (i.e., domain-dependent) queries that may potentially produce infinite evaluation results.

To answer RQ1, we compare our tool with the translation-based approach by Van Gelder and Topor [GT91] (VGT), the structure reduction approach by Ailamazyan et al. [AGSS86], and the DDD [MLAH99, Møl02], LDD [CGS09], and MonPoly$^{\mathsf{REG}}$ [BKMZ15] tools that evaluate RC queries directly using infinite relations encoded as binary decision diagrams.

Experiments:	SMALL	MEDIUM	LARGE	INFINITE	REAL
RC2SQL	✓	✓	✓	✓*	✓
VGT	✓	✓	TO	N/A	✓
DDD	✓	TO	TO	✓	✓
LDD	✓	TO	TO	✓	✓
MonPoly$^{\mathsf{REG}}$	✓	TO	TO	✓	✓
Ailamazyan et al.	TO	TO	TO	TO	TO

∗ Only states that the result is infinite.

Table 1: Applicability and performance of all the tools on all the experiments. TO = Timeout of 300s on all experiment runs, N/A = Not applicable

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 Figure 7 to use the relation $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{A})$ (Appendix A, Figure 17) instead of $\mathsf{cov}(x, Q, \mathcal{G})$ (Figure 5) and to use the generator $\bigvee_{Q_{ap} \in \mathcal{A}} \exists \vec{\mathsf{fv}}(Q) \setminus \{x\}.\, Q_{ap}$ instead of $\mathsf{qps}^{\vee}(\mathcal{G})$. Evaluable queries Q are always translated into $(Q_{fin}, \bot)$ by $\mathsf{rw}(\cdot)$ because all of Q's free variables are range restricted. We exclude VGT from the comparison on non-evaluable queries (experiment INFINITE). Similarly, the implementation of Ailamazyan et al.'s approach was not available; hence we used our formally-verified implementation [Ras22]. The implementations of the remaining tools were publicly available.

We use Data Golf structures of growing size (experiments SMALL, MEDIUM, and LARGE) to answer RQ2. In contrast, to answer RQ3, we use real-world structures obtained from the Amazon review dataset [NLM19] (experiment REAL).

To answer RQ4, we also consider variants of the translation-based approaches without the step that uses count aggregation optimization $\mathsf{optcnt}(\cdot)$, superscripted with a minus ($^-$).

SQL queries computed by the translations are evaluated using the PostgreSQL and MySQL DBMS (RQ5). We superscript the tool names with $^{\mathsf{P}}$ and $^{\mathsf{M}}$ accordingly. In the LARGE experiment, we only use PostgreSQL because it consistently performed better than MySQL in the MEDIUM experiment. In all our experiments, the translation-based tools used a Data Golf structure with $|\mathcal{T}^+| = |\mathcal{T}^-| = 2$ as the training database. We run our experiments on an AMD Ryzen 7 PRO 4750U computer with 32 GB RAM. The relations in PostgreSQL and MySQL are recreated before each invocation to prevent optimizations based on caching recent query evaluation results. We measure the query evaluation times of all the tools and the translation time of our RC2SQL tool (RQ6). We provide all our experiments in an easily reproducible and publicly available artifact [RBKT22a].

In the SMALL, MEDIUM, and LARGE experiments, we generate ten pseudorandom queries (denoted as $Q_i, 1 \leq i \leq 10$, see Appendix C) with a fixed size 14 and Data Golf structures $\mathcal{S}$ (strategy $\gamma = 1$). The queries satisfy the Data Golf assumptions along with a few additional ones: the queries are not safe range, every bound variable actually occurs in its scope, disjunction only appears at the top-level, 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 15.

The INFINITE experiment consists of five pseudorandom queries $Q_i^I, 1 \leq i \leq 5$ (Appendix C) that are *not* evaluable and $\mathsf{rw}(Q_i^I) = (Q_{i,fin}, Q_{i,inf})$, where $Q_{i,inf} \neq \bot$. 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. We choose the queries so that the number of their satisfying tuples is not too high, e.g., quadratic in the parameter n, because no tool can possibly enumerate so many tuples within the timeout. For each $1 \leq i \leq 5$, we compare the performance of our tool to tools that directly evaluate Q_i^I 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 $Q_{i,inf}$), whereas the other tools also output a finite representation of the infinite result. For finite results, all tools produce the same output.

Figure 15 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. In the experiments SMALL, MEDIUM, and LARGE, the columns correspond to ten unique pseudorandom queries (the same queries are used in all the three experiments). In the INFINITE experiment, we use five unique pseudorandom queries and two Data Golf strategies. The time it takes for our translation RC2SQL to translate each query is shown in the first line for the experiments SMALL and INFINITE because the queries in the experiments MEDIUM and LARGE are the same as in SMALL. The remaining lines show evaluation times with the lowest time for a query typeset in bold. We omit the rows for tools that time out or crash on all queries of an experiment, e.g., Ailamazyan et al. [AGSS86]. Table 1 summarizes which tools were used in which experiment. We conclude that our translation RC2SQL significantly outperforms all other tools on all queries (RQ1) (except VGT on the fourth query, but on the smallest structure) and scales well to higher values of n, i.e., larger relations in the Data Golf structures, on all queries (RQ2). The count aggregation optimization (RQ4) provides no conclusive benefit for our translation, while it consistently improves VGT's performance, especially in the MEDIUM experiment.

We also evaluate the tools on the queries Q^{susp} and Q^{susp}_{user} from the introduction and on the more challenging query $Q^{susp}_{text} \coloneqq \mathsf{B}(b) \wedge \exists u, s, t.\, \forall p.\, \mathsf{P}(b, p) \longrightarrow \mathsf{S}(p, u, s) \vee \mathsf{T}(p, u, t)$ 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 [NLM19]. The real-world relations P, S, and T are obtained by projecting the respective tables from the Amazon review dataset for two chosen product categories: gift cards (abbreviated GC) consisting of 147 194 reviews of 1 548 products and musical instruments (MI) consisting of 1 512 530 reviews of 120 400 products. 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 16 shows the empirical evaluation results: the time it takes for our translation RC2SQL to translate each query is shown in the first line and the execution times on Data Golf structures (left) and on structures derived from the real-world dataset for two specific product categories (right) are shown in the remaining lines. We remark that VGT cannot handle the query Q^{susp}_{user} as it is not evaluable [GT91], hence we mark the correspond cells in Figure 16 with $-$. Our translation RC2SQL significantly outperforms all other tools (except VGT on Q^{susp}, where RC2SQL and VGT have similar performance) on both Data Golf and

Query	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}
Experiment SMALL, Evaluable pseudorandom queries Q, $\|\mathsf{sub}(Q)\| = 14$, $n = 500$:										
Translation time	0.6	0.0	0.1	0.0	0.0	0.0	0.0	0.1	0.0	0.0
RC2SQL$^{\mathsf{P}}$	**0.2**	0.2	0.3	0.2	0.2	**0.2**	0.2	**0.1**	**0.2**	**0.1**
RC2SQL$^{\mathsf{M}}$	0.3	0.2	0.3	0.2	0.2	**0.2**	**0.1**	0.2	0.3	0.2
RC2SQL$^{-\mathsf{P}}$	**0.2**	**0.1**	**0.2**	0.2	**0.1**	**0.2**	**0.1**	**0.1**	**0.2**	**0.1**
RC2SQL$^{-\mathsf{M}}$	**0.2**	**0.1**	0.3	0.2	**0.1**	**0.2**	**0.1**	**0.1**	0.4	0.3
VGT$^{\mathsf{P}}$	1.5	0.2	1.8	1.3	1.9	8.6	2.5	1.5	15.6	4.8
VGT$^{\mathsf{M}}$	0.3	0.2	0.3	**0.1**	0.2	56.3	6.1	0.2	155.7	13.5
VGT$^{-\mathsf{P}}$	16.4	6.0	10.7	6.3	TO	6.2	3.1	2.3	48.0	8.8
VGT$^{-\mathsf{M}}$	129.1	73.7	97.3	66.8	TO	52.1	19.1	12.8	TO	61.1
DDD	3.8	2.4	5.5	RE	1.2	3.7	4.7	2.2	17.9	5.3
LDD	35.9	16.0	46.2	15.6	9.0	28.3	12.9	17.4	206.0	32.6
MonPoly$^{\mathsf{REG}}$	31.3	11.2	29.7	10.5	10.2	31.2	10.7	21.0	103.3	24.0
Experiment MEDIUM, Evaluable pseudorandom queries Q, $\|\mathsf{sub}(Q)\| = 14$, $n = 20000$:										
RC2SQL$^{\mathsf{P}}$	2.1	1.1	2.2	1.2	1.1	1.2	**0.5**	0.9	1.8	**1.0**
RC2SQL$^{\mathsf{M}}$	6.2	2.6	6.6	2.7	2.4	4.4	1.6	2.7	9.0	3.3
RC2SQL$^{-\mathsf{P}}$	**1.4**	**0.8**	**1.4**	**0.8**	**0.8**	**1.0**	**0.5**	**0.6**	**1.7**	**1.0**
RC2SQL$^{-\mathsf{M}}$	4.3	1.9	5.0	1.8	1.9	3.3	1.6	2.2	8.3	3.2
VGT$^{\mathsf{P}}$	2.9	1.0	3.0	2.2	2.8	TO	TO	2.3	TO	TO
VGT$^{\mathsf{M}}$	4.6	2.4	5.2	2.6	2.7	TO	TO	3.0	TO	TO
VGT$^{-\mathsf{P}}$	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
VGT$^{-\mathsf{M}}$	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Experiment LARGE, Evaluable pseudorandom queries Q, $\|\mathsf{sub}(Q)\| = 14$, tool = RC2SQL$^{\mathsf{P}}$:										
$n = 40000$	4.5	2.3	4.4	2.3	2.2	2.5	1.0	1.7	3.7	1.8
$n = 80000$	8.8	4.5	8.7	4.8	4.5	5.0	1.8	3.3	7.2	3.7
$n = 120000$	14.1	6.8	12.8	7.2	7.0	7.1	2.8	5.1	10.8	4.9
Query	Q_1^I	Q_2^I	Q_3^I	Q_4^I	Q_5^I	Q_1^I	Q_2^I	Q_3^I	Q_4^I	Q_5^I
Experiment INFINITE, Non-evaluable pseudorandom queries Q, $\|\mathsf{sub}(Q)\| = 7$, $n = 4000$:										
	Infinite results ($\gamma = 0$)					Finite results ($\gamma = 1$)				
Translation time	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RC2SQL$^{\mathsf{P}}$	0.5	0.5	0.5	0.5	0.5	0.5	1.5	0.7	0.7	1.4
RC2SQL$^{\mathsf{M}}$	**0.3**	0.5	**0.3**	0.5	0.5	**0.4**	**0.7**	0.4	0.6	**0.6**
RC2SQL$^{-\mathsf{P}}$	**0.3**	**0.3**	**0.3**	**0.3**	**0.3**	**0.4**	TO	**0.3**	**0.5**	TO
RC2SQL$^{-\mathsf{M}}$	0.4	1.0	0.5	0.7	0.9	0.6	TO	0.6	0.6	TO
DDD	32.4	81.0	32.7	60.9	81.6	32.1	68.6	31.9	59.4	68.2
LDD	TO	TO	TO	TO	TO	288.5	TO	TO	TO	TO
MonPoly$^{\mathsf{REG}}$	175.0	TO	175.4	TO	TO	160.3	299.1	160.0	TO	TO

Figure 15: Experiments SMALL, MEDIUM, LARGE, and INFINITE. We use the following abbreviations: TO = Timeout of 300s, RE = Runtime Error. Reported translation time for RC2SQL.

Query	Q^{susp}		Q^{susp}_{user}		Q^{susp}_{text}		Q^{susp}		Q^{susp}_{user}		Q^{susp}_{text}	
Param. n	10^3	10^4	10^3	10^4	10^3	10^4	Dataset GC	MI	GC	MI	GC	MI
Translation time	0.0		0.0		0.3		0.0		0.0		0.3	
RC2SQL$^{\mathsf{P}}$	1.2	1.4	1.8	2.2	3.5	4.1	**1.6**	**8.4**	2.5	**11.9**	4.9	**51.3**
RC2SQL$^{\mathsf{M}}$	**0.3**	**1.1**	**0.3**	**1.4**	**0.5**	**2.6**	1.9	54.2	**2.4**	71.0	**4.5**	TO
RC2SQL$^{-\mathsf{P}}$	28.1	TO	28.6	TO	228.6	TO	132.9	TO	131.8	TO	TO	TO
RC2SQL$^{-\mathsf{M}}$	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
VGT$^{\mathsf{P}}$	1.5	1.7	–	–	204.3	TO	1.9	10.0	–	–	TO	TO
VGT$^{\mathsf{M}}$	**0.3**	1.4	–	–	TO	TO	1.7	60.9	–	–	TO	TO
VGT$^{-\mathsf{P}}$	TO	TO	–	–	TO	TO	TO	TO	–	–	TO	TO
VGT$^{-\mathsf{M}}$	TO	TO	–	–	TO	TO	TO	TO	–	–	TO	TO
DDD	4.0	TO	4.1	TO	18.6	TO	61.3	TO	61.0	TO	126.7	TO
LDD	22.3	TO	22.2	TO	148.5	TO	TO	TO	TO	TO	TO	TO
MonPoly$^{\mathsf{REG}}$	22.4	TO	22.6	TO	84.1	TO	TO	TO	TO	TO	TO	TO

Figure 16: Experiment Real with the queries Q^{susp}, Q^{susp}_{user}, Q^{susp}_{text}. We use the following abbreviations: GC = Gift Cards, MI = Musical Instruments, TO = Timeout of 300s. Reported translation time for RC2SQL.

real-world structures (RQ3). VGT^{-} translates Q^{susp} into a RANF query with a higher query cost than RC2SQL^{-}. However, the optimization optcnt$(\cdot)$ manages to rectify this inefficiency (RQ4) 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 $n = 20$ [RBKT22a]. Nevertheless, VGT does not finish evaluating the query Q^{susp}_{text} on GC and MI datasets within 5 minutes, unlike RC2SQL. Finally, RC2SQL's translation took less than 1 second on all the queries (RQ6).

9. 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, and aggregations.

Acknowledgments. This research has been supported by the Swiss National Science Foundation grants "Big Data Monitoring" (167162) and "Model-driven Security & Privacy" (204796) as well as by a Novo Nordisk Fonden start package grant (NNF20OC0063462).

References

[AGSS86] 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`.

[AH91] Arnon Avron and Yoram Hirshfeld. On first order database query languages. In *LICS 1991*, pages 226–231. IEEE Computer Society, 1991. `doi:10.1109/LICS.1991.151647`.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: http://webdam.inria.fr/Alice/.

[BG04] 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.

[BKMZ15] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015. doi:10.1145/2699444.

[BL00] Michael Benedikt and Leonid Libkin. Relational queries over interpreted structures. *J. ACM*, 47(4):644–680, 2000. doi:10.1145/347476.347477.

[CGS09] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Decision diagrams for linear arithmetic. In *FMCAD 2009*, pages 53–60. IEEE, 2009. doi:10.1109/FMCAD.2009.5351143.

[CKMP97] 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 1997*, pages 286–295. Morgan Kaufmann, 1997. URL: http://www.vldb.org/conf/1997/P286.PDF.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. *Research Report / RJ / IBM / San Jose, California*, RJ987, 1972.

[CT95] 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.

[EHJ93] Martha Escobar-Molano, Richard Hull, and Dean Jacobs. Safety and translation of calculus queries with scalar functions. In Catriel Beeri, editor, *PODS 1993*, pages 253–264. ACM Press, 1993. doi:10.1145/153850.153909.

[Ell13] Erling Ellingsen. Regex golf, 2013. URL: https://alf.nu/RegexGolf.

[GT87] Allen Van Gelder and Rodney W. Topor. Safety and correct translation of relational calculus formulas. In Moshe Y. Vardi, editor, *PODS 1987*, pages 313–327. ACM, 1987. doi:10.1145/28659.28693.

[GT91] 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.

[HS94] Richard Hull and Jianwen Su. Domain independence and the relational calculus. *Acta Informatica*, 31(6):513–524, 1994. doi:10.1007/BF01213204.

[Kif88] Michael Kifer. On safety, domain independence, and capturability of database queries (preliminary report). In Catriel Beeri, Joachim W. Schmidt, and Umeshwar Dayal, editors, *JCDKB 1988*, pages 405–415. Morgan Kaufmann, 1988. doi:10.1016/b978-1-4832-1313-2.50037-8.

[KM01] Nils Klarlund and Anders Møller. *MONA v1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, 2001. URL: http://www.brics.dk/mona/.

[KNR16] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. In Tova Milo and Wang-Chiew Tan, editors, *PODS 2016*, pages 13–28. ACM, 2016. doi:10.1145/2902251.2902280.

[Lib04] 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.

[LYL08] 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.

[MLAH99] Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Difference decision diagrams. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *CSL 1999*, volume 1683 of *LNCS*, pages 111–125. Springer, 1999. doi:10.1007/3-540-48168-0_9.

[Møl02] Jesper B. Møller. DDDLIB: A library for solving quantified difference inequalities. In Andrei Voronkov, editor, *CADE 2002*, volume 2392 of *LNCS*, pages 129–133. Springer, 2002. doi:10.1007/3-540-45620-1_9.

[NLM19] 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 2019*, pages 188–197. Association for Computational Linguistics, 2019. doi:10.18653/v1/D19-1018.

[NRR13] 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.

[OS16] Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Rec.*, 45(2):5–16, 2016. doi:10.1145/3003665.3003667.

[Pao69] 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.

[Ras22] Martin Raszyk. First-order query evaluation. *Archive of Formal Proofs*, 2022. Formal proof development. URL: https://www.isa-afp.org/entries/Eval_FO.html.

[RBKT22a] Martin Raszyk, David Basin, Srđan Krstić, and Dmitriy Traytel. Implementation and empirical evaluation associated with this article, 2022. URL: https://github.com/mraszyk/rc2sql/tree/lmcs22.

[RBKT22b] Martin Raszyk, David Basin, Srđan Krstić, and Dmitriy Traytel. Practical relational calculus query evaluation. In Dan Olteanu and Nils Vortmeier, editors, *ICDT 2022*, volume 220 of *LIPIcs*, pages 11:1–11:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICDT.2022.11.

[Rev02] Peter Z. Revesz. *Introduction to Constraint Databases*. Texts in Computer Science. Springer, 2002. doi:10.1007/b97430.

[RT22] Martin Raszyk and Dmitriy Traytel. Making arbitrary relational calculus queries safe-range. *Archive of Formal Proofs*, 2022. Formal proof development. URL: https://www.isa-afp.org/entries/Safe_Range_RC.html.

[Tra50] Boris A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70(4):569–572, 1950.

[Var81] 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.

[Var82] 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 1982*, pages 137–146. ACM, 1982. doi:10.1145/800070.802186.

[Yan19] Jun Yang. radb, 2019. URL: https://github.com/junyang/radb.

$$
\begin{array}{ll}
\mathsf{gen}_{\mathsf{vgt}}(x, Q, \{Q\}) & \text{if } \mathsf{ap}(Q) \text{ and } x \in \mathsf{fv}(Q); \\
\mathsf{gen}_{\mathsf{vgt}}(x, \neg\neg Q, \mathcal{A}) & \text{if } \mathsf{gen}_{\mathsf{vgt}}(x, Q, \mathcal{A}); \\
\mathsf{gen}_{\mathsf{vgt}}(x, \neg(Q_1 \vee Q_2), \mathcal{A}) & \text{if } \mathsf{gen}_{\mathsf{vgt}}(x, (\neg Q_1) \wedge (\neg Q_2), \mathcal{A}); \\
\mathsf{gen}_{\mathsf{vgt}}(x, \neg(Q_1 \wedge Q_2), \mathcal{A}) & \text{if } \mathsf{gen}_{\mathsf{vgt}}(x, (\neg Q_1) \vee (\neg Q_2), \mathcal{A}); \\
\mathsf{gen}_{\mathsf{vgt}}(x, \neg\exists y.\, Q_y, \mathcal{A}) & \text{if } x \neq y \text{ and } \mathsf{gen}_{\mathsf{vgt}}(x, \neg Q_y, \mathcal{A}); \\
\mathsf{gen}_{\mathsf{vgt}}(x, Q_1 \vee Q_2, \mathcal{A}_1 \cup \mathcal{A}_2) & \text{if } \mathsf{gen}_{\mathsf{vgt}}(x, Q_1, \mathcal{A}_1) \text{ and } \mathsf{gen}_{\mathsf{vgt}}(x, Q_2, \mathcal{A}_2); \\
\mathsf{gen}_{\mathsf{vgt}}(x, Q_1 \wedge Q_2, \mathcal{A}) & \text{if } \mathsf{gen}_{\mathsf{vgt}}(x, Q_1, \mathcal{A}); \\
\mathsf{gen}_{\mathsf{vgt}}(x, Q_1 \wedge Q_2, \mathcal{A}) & \text{if } \mathsf{gen}_{\mathsf{vgt}}(x, Q_2, \mathcal{A}); \\
\mathsf{gen}_{\mathsf{vgt}}(x, \exists y.\, Q_y, \mathcal{A}) & \text{if } x \neq y \text{ and } \mathsf{gen}_{\mathsf{vgt}}(x, Q_y, \mathcal{A}); \\
 & \\
\mathsf{con}_{\mathsf{vgt}}(x, Q, \varnothing) & \text{if } x \notin \mathsf{fv}(Q); \\
\mathsf{con}_{\mathsf{vgt}}(x, Q, \{Q\}) & \text{if } \mathsf{ap}(Q) \text{ and } x \in \mathsf{fv}(Q); \\
\mathsf{con}_{\mathsf{vgt}}(x, \neg\neg Q, \mathcal{A}) & \text{if } \mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{A}); \\
\mathsf{con}_{\mathsf{vgt}}(x, \neg(Q_1 \vee Q_2), \mathcal{A}) & \text{if } \mathsf{con}_{\mathsf{vgt}}(x, (\neg Q_1) \text{ and } (\neg Q_2), \mathcal{A}); \\
\mathsf{con}_{\mathsf{vgt}}(x, \neg(Q_1 \wedge Q_2), \mathcal{A}) & \text{if } \mathsf{con}_{\mathsf{vgt}}(x, (\neg Q_1) \vee (\neg Q_2), \mathcal{A}); \\
\mathsf{con}_{\mathsf{vgt}}(x, \neg\exists y.\, Q_y, \mathcal{A}) & \text{if } x \neq y \text{ and } \mathsf{con}_{\mathsf{vgt}}(x, \neg Q_y, \mathcal{A}); \\
\mathsf{con}_{\mathsf{vgt}}(x, Q_1 \vee Q_2, \mathcal{A}_1 \cup \mathcal{A}_2) & \text{if } \mathsf{con}_{\mathsf{vgt}}(x, Q_1, \mathcal{A}_1) \text{ and } \mathsf{con}_{\mathsf{vgt}}(x, Q_2, \mathcal{A}_2); \\
\mathsf{con}_{\mathsf{vgt}}(x, Q_1 \wedge Q_2, \mathcal{A}) & \text{if } \mathsf{gen}_{\mathsf{vgt}}(x, Q_1, \mathcal{A}); \\
\mathsf{con}_{\mathsf{vgt}}(x, Q_1 \wedge Q_2, \mathcal{A}) & \text{if } \mathsf{gen}_{\mathsf{vgt}}(x, Q_2, \mathcal{A}); \\
\mathsf{con}_{\mathsf{vgt}}(x, Q_1 \wedge Q_2, \mathcal{A}_1 \cup \mathcal{A}_2) & \text{if } \mathsf{con}_{\mathsf{vgt}}(x, Q_1, \mathcal{A}_1) \text{ and } \mathsf{con}_{\mathsf{vgt}}(x, Q_2, \mathcal{A}_2); \\
\mathsf{con}_{\mathsf{vgt}}(x, \exists y.\, Q_y, \mathcal{A}) & \text{if } x \neq y \text{ and } \mathsf{con}_{\mathsf{vgt}}(x, Q_y, \mathcal{A}).
\end{array}
$$

Figure 17: The relations $\mathsf{gen}_{\mathsf{vgt}}(x, Q, \mathcal{A})$ and $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{A})$ [GT91].

Appendix A. Evaluable Queries

The classes of *evaluable* queries [GT91, Definition 5.2] and *allowed* queries [GT91, Definition 5.3] are decidable subsets of domain-independent RC queries. The evaluable queries characterize exactly the domain-independent queries with no repeated predicate symbols [GT91, Theorem 10.5]. Every evaluable query can be translated to an equivalent allowed query [GT91, Theorem 8.6] and every allowed query can be translated to an equivalent RANF query [GT91, Theorem 9.6].

Definition A.1. *A query Q is called* evaluable *if*

- *every variable $x \in \mathsf{fv}(Q)$ satisfies $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ and*
- *the bound variable y in every subquery $\exists y.\, Q_y$ of Q satisfies $\mathsf{con}_{\mathsf{vgt}}(y, Q_y)$.*

A query Q is called allowed *if*

- *every variable $x \in \mathsf{fv}(Q)$ satisfies $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ and*
- *the bound variable y in every subquery $\exists y.\, Q_y$ of Q satisfies $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$,*

where the relation $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ is defined to hold iff there exists a set of atomic predicates $\mathcal{A}$ such that $\mathsf{gen}_{\mathsf{vgt}}(x, Q, \mathcal{A})$ and the relation $\mathsf{con}_{\mathsf{vgt}}(x, Q)$ is defined to hold iff there exists a set of atomic predicates $\mathcal{A}$ such that $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{A})$, respectively. The relations $\mathsf{gen}_{\mathsf{vgt}}(x, Q, \mathcal{A})$ and $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{A})$ are defined in Figure 17.

The termination of the rules in Figure 17 follow using the measure $\mathsf{m}(Q)$ (Figure 11). We now relate the definitions from Figure 4 and Figure 17 with the following lemmas.

Lemma A.2. *Let x and y be free variables in a query Q such that $\mathsf{gen}_{\mathsf{vgt}}(x, \neg Q)$ and $\mathsf{gen}_{\mathsf{vgt}}(y, Q)$ hold. Then we get a contradiction.*

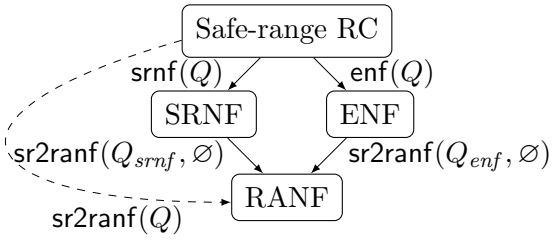

Figure 18: Alternative translation from safe-range RC to RANF query via ENF.

Proof. This is proved by induction on the query Q using the measure $\mathsf{m}(Q)$ on queries defined in Figure 11, which decreases in every case of the definition in Figure 17. □

Lemma A.3. *Let Q be a query such that $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$ holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q. Suppose that $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ holds for a free variable $x \in \mathsf{fv}(Q)$. Then $\mathsf{gen}(x, Q)$ holds.*

Proof. This is proved by induction on the query Q using the measure $\mathsf{m}(Q)$ on queries defined in Figure 11, which decreases in every case of the definition in Figure 17.

Lemma A.2 and the assumption that $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$ holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q imply that $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ cannot be derived using the rule $\mathsf{gen}_{\mathsf{vgt}}(x, \neg\exists y.\, Q_y)$, i.e., Q cannot be of the form $\neg\exists y.\, Q_y$. Every other case in the definition of $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ has a corresponding case in the definition of $\mathsf{gen}(x, Q)$. □

Lemma A.4. *Let Q be an* allowed *query, i.e., $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ holds for every free variable $x \in \mathsf{fv}(Q)$ and $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$ holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q. Then Q is a safe-range query, i.e., $\mathsf{gen}(x, Q)$ holds for every free variable $x \in \mathsf{fv}(Q)$ and $\mathsf{gen}(y, Q_y)$ holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q.*

Proof. The lemma is proved by applying Lemma A.3 to every free variable of Q and to the bound variable y in every subquery of Q of the form $\exists y.\, Q_y$. □

Lemma A.4 shows that every allowed query is safe range. But there exist safe-range queries that are not allowed, e.g., $\mathsf{B}(x) \wedge x \approx y$.

Appendix B. Existential Normal Form

Recall that our translation uses a standard approach to obtain RANF queries from safe-range queries via SRNF [AHV95]. In this section we introduce existential normal form (ENF), an alternative normal form to SRNF that can also be used to translate safe-range queries to RANF queries, and we discuss why we opt for using SRNF instead.

Figure 18 shows an overview of the RC fragments and query normal forms (nodes) and the functions we use to translate between them (edges). The dashed edge shows the translation of a safe-range query to RANF we opt for in this article. It is the composition of the two translations from safe-range RC to SRNF and from SRNF to RANF, respectively. In the rest of this section we introduce ENF and the corresponding translations to RANF.

ENF was introduced by Van Gelder and Topor [GT91] to translate an allowed query [GT91] into an equivalent RANF query. Given a safe-range query in ENF, the rules $(R1)$–$(R3)$ from Section 3.4 can be applied to obtain an equivalent RANF query [EHJ93, Lemma 7.8].

We remark that the rules $(R1)$–$(R3)$ are not sufficient to yield an equivalent RANF query for the original definition of ENF [GT91]. This issue has been identified and fixed by Escobar-Molano et al. [EHJ93]. Unlike SRNF, a query in ENF can have a subquery of the form $\neg(Q_1 \wedge Q_2)$, but no subquery of the form $\neg Q_1 \vee Q_2$ or $Q_1 \vee \neg Q_2$. A function $\mathsf{enf}(Q)$ that yields an ENF query equivalent to Q can be defined in terms of subquery rewriting using the rules in [EHJ93, Figure 2].

Analogously to [EHJ93, Lemma 7.4], if a query Q is safe range, then $\mathsf{enf}(Q)$ is also safe range. Next we prove the following lemma that we could use as a precondition for translating safe-range queries in ENF to queries in RANF.

Lemma B.1. *Let Q_{enf} be a query in ENF. Then $\mathsf{gen}(x, \neg Q')$ does not hold for any variable x and subquery $\neg Q'$ of Q_{enf}.*

Proof. Assume that $\mathsf{gen}(x, \neg Q')$ holds for a variable x in a subquery $\neg Q'$ of Q_{enf}. We derive a contradiction by induction on $\mathsf{m}(Q_{enf})$. According to Figure 4 and by definition of ENF, $\mathsf{gen}(x, \neg Q')$ can only hold if Q' is a conjunction. Then $\mathsf{gen}(x, \neg Q')$ implies $\mathsf{gen}(x, \neg Q_1)$ for some query $Q_1 \in \mathsf{flat}^{\wedge}(Q')$ that is not a negation (by definition of ENF) or conjunction (by definition of $\mathsf{flat}^{\wedge}(\cdot)$), i.e., Q_1 is a disjunction (according to Figure 4). Then $\mathsf{gen}(x, \neg Q_1)$ implies $\mathsf{gen}(x, \neg Q_2)$ for some query $Q_2 \in \mathsf{flat}^{\vee}(Q_1)$ that is not a negation (by definition of ENF) or disjunction (by definition of $\mathsf{flat}^{\vee}(\cdot)$), i.e., Q_2 is a conjunction (according to Figure 4). Next we observe that $\neg Q_2$ is in ENF because Q_2 is a subquery of the ENF query Q_{enf}, Q_2 is a conjunction, and Q_2 is a subquery of a disjunction (Q_1) in Q_{enf}. Moreover, $\mathsf{m}(\neg Q_2) < \mathsf{m}(Q_1) < \mathsf{m}(Q') < \mathsf{m}(Q_{enf})$. This allows us to apply the induction hypothesis to the ENF query $\neg Q_2$ and its subquery $\neg Q_2$ (note that a query is a subquery of itself) and derive that $\mathsf{gen}(x, \neg Q_2)$ does not hold, which is a contradiction. □

Although applying the rules $(R1)$–$(R3)$ to $\mathsf{enf}(Q)$ instead of $\mathsf{srnf}(Q)$ may result in a RANF query with fewer subqueries, the query cost, i.e., the time complexity of query evaluation, can be arbitrarily larger. We illustrate this in the following example that is also included in our artifact [RBKT22a]. We thus opt for using SRNF instead of ENF for translating safe-range queries into RANF.

Example B.2. *The safe-range query $Q_{enf} \coloneqq \mathsf{P}_2(x, y) \wedge \neg(\mathsf{P}_1(x) \wedge \mathsf{P}_1(y))$ is in ENF and RANF, but not SRNF. Applying the rule $(R1)$ to $\mathsf{srnf}(Q_{enf})$ yields the RANF query $Q_{srnf} \coloneqq (\mathsf{P}_2(x, y) \wedge \neg\mathsf{P}_1(x)) \vee (\mathsf{P}_2(x, y) \wedge \neg\mathsf{P}_1(y))$ that is equivalent to Q_{enf}. The costs of the two queries over a structure $\mathcal{S}$ are $\mathsf{cost}^{\mathcal{S}}(Q_{enf}) = 2 \cdot |[\![\mathsf{P}_2(x, y)]\!]| + |[\![\mathsf{P}_1(x)]\!]| + |[\![\mathsf{P}_1(y)]\!]| + 2 \cdot |[\![\mathsf{P}_1(x) \wedge \mathsf{P}_1(y)]\!]| + 2 \cdot |[\![Q_{enf}]\!]|$ and $\mathsf{cost}^{\mathcal{S}}(Q_{srnf}) = 2 \cdot |[\![\mathsf{P}_2(x, y)]\!]| + |[\![\mathsf{P}_1(x)]\!]| + 2 \cdot |[\![\mathsf{P}_2(x, y)]\!]| + |[\![\mathsf{P}_1(y)]\!]| + 2 \cdot |[\![\mathsf{P}_2(x, y) \wedge \neg\mathsf{P}_1(x)]\!]| + 2 \cdot |[\![\mathsf{P}_2(x, y) \wedge \neg\mathsf{P}_1(y)]\!]| + 2 \cdot |[\![Q_{srnf}]\!]|$, respectively. Note that the cost of Q_{enf} can be arbitrarily larger if $\mathsf{P}_1(x) \wedge \mathsf{P}_1(y)$ evaluates to a large intermediate result, i.e., $|[\![\mathsf{P}_1(x) \wedge \mathsf{P}_1(y)]\!]| \gg |[\![\mathsf{P}_2(x, y)]\!]|$. In contrast, the cost of Q_{srnf} can only be larger by a constant factor.*

Appendix C. Queries used in the evaluation

Figure 19 shows the randomly generated queries that we use in our evaluation. The index of each predicate indicates the predicate's arity.

$$
\begin{aligned}
Q_1 &\coloneqq \neg(\exists x_2.\, \neg(\exists x_3.\, ((\mathsf{P}_3(x_1, x_0, x_3)) \wedge (\neg(\exists x_4.\, \mathsf{P}_4(x_1, x_3, x_4, x_2)))) \wedge (\exists x_4.\, \mathsf{P}_2(x_1, x_4)))) \vee (\mathsf{Q}_2(x_1, x_0))\\
Q_2 &\coloneqq \neg(\exists x_2.\, \neg(\exists x_3.\, (\neg(\mathsf{P}_4(x_1, x_0, x_2, x_3))) \wedge (\mathsf{P}_3(x_1, x_3, x_0)))) \wedge (\exists x_2.\, \exists x_3.\, (x_1 = x_2) \wedge (\mathsf{Q}_3(x_0, x_2, x_3)))\\
Q_3 &\coloneqq (\exists x_2.\, \neg(\exists x_3.\, ((\mathsf{P}_3(x_1, x_0, x_3)) \wedge (\mathsf{P}_1(x_0))) \wedge ((x_0 = x_1) \wedge (\neg(\mathsf{P}_4(x_1, x_2, x_3, x_0)))))) \wedge (\exists x_2.\, \mathsf{Q}_3(x_1, x_2, x_0))\\
Q_4 &\coloneqq (\exists x_2.\, \mathsf{P}_3(x_1, x_2, x_0)) \wedge (\neg(x_0 = x_1)) \wedge (\neg(\exists x_2.\, \neg(\exists x_3.\, (\mathsf{P}_2(x_0, x_3)) \wedge (\neg(\mathsf{P}_4(x_1, x_3, x_2, x_0))))))\\
Q_5 &\coloneqq (\exists x_2.\, \exists x_3.\, \neg(\exists x_4.\, (\exists x_5.\, \mathsf{P}_3(x_0, x_5, x_4)) \wedge (\neg(\mathsf{P}_4(x_0, x_4, x_2, x_3))))) \wedge ((\neg(x_0 = x_1)) \wedge (\mathsf{P}_2(x_0, x_1)))\\
Q_6 &\coloneqq (\exists x_2.\, \neg(\exists x_3.\, ((\neg(\exists x_4.\, \mathsf{P}_3(x_0, x_3, x_4))) \wedge (\mathsf{Q}_3(x_0, x_1, x_3))) \wedge (\neg(\mathsf{R}_3(x_0, x_2, x_1))))) \wedge (\exists x_2.\, \mathsf{S}_3(x_0, x_1, x_2))\\
Q_7 &\coloneqq (\exists x_2.\, (\mathsf{P}_3(x_0, x_2, x_1)) \wedge (x_0 = x_2)) \wedge (\exists x_2.\, \neg(\exists x_3.\, (\neg(\exists x_4.\, \exists x_5.\, \mathsf{P}_4(x_0, x_2, x_5, x_4))) \wedge (\mathsf{P}_2(x_1, x_3))))\\
Q_8 &\coloneqq \neg(\exists x_2.\, \neg(\exists x_3.\, (\exists x_4.\, (\mathsf{P}_3(x_0, x_3, x_1)) \wedge ((\mathsf{P}_4(x_0, x_3, x_1, x_4)) \wedge (x_3 = x_4))) \wedge (\neg(\exists x_4.\, \mathsf{Q}_4(x_0, x_4, x_3, x_2)))))\\
Q_9 &\coloneqq \mathsf{P}_2(x_1, x_0) \wedge (\exists x_2.\, \neg(\exists x_3.\, ((\neg(\mathsf{P}_4(x_1, x_3, x_2, x_0))) \wedge (\mathsf{P}_3(x_0, x_3, x_1))) \wedge (\neg(\mathsf{Q}_2(x_0, x_3))))) \wedge (x_0 = x_1)\\
Q_{10} &\coloneqq (\exists x_2.\, \mathsf{P}_3(x_1, x_2, x_0)) \vee (\neg(\exists x_2.\, \neg(\exists x_3.\, (\neg(\mathsf{P}_2(x_1, x_2))) \wedge (\exists x_4.\, (\mathsf{P}_1(x_0)) \wedge (\mathsf{P}_4(x_0, x_3, x_1, x_4))))))\\
Q_1^I &\coloneqq \mathsf{P}_1(x_0) \wedge \neg(\exists x_2.\, \exists x_3.\, (\mathsf{P}_3(x_0, x_2, x_3) \wedge \neg(x_0 = x_1)))\\
Q_2^I &\coloneqq \mathsf{P}_1(x_0) \wedge \neg(\exists x_2.\, \exists x_3.\, (\mathsf{P}_3(x_0, x_2, x_3) \wedge \neg\mathsf{P}_2(x_1, x_3)))\\
Q_3^I &\coloneqq \mathsf{P}_1(x_0) \wedge \neg(\exists x_2.\, \exists x_3.\, (\mathsf{P}_3(x_0, x_3, x_2) \wedge \neg(x_1 = x_2)))\\
Q_4^I &\coloneqq \mathsf{P}_1(x_1) \wedge \neg(\exists x_2.\, \exists x_3.\, (\mathsf{P}_3(x_1, x_3, x_2) \wedge \neg\mathsf{P}_4(x_1, x_0, x_3, x_2)))\\
Q_5^I &\coloneqq \mathsf{P}_1(x_0) \wedge \neg(\exists x_2.\, \exists x_3.\, (\mathsf{P}_3(x_0, x_2, x_3) \wedge \neg\mathsf{Q}_3(x_1, x_2, x_3)))
\end{aligned}
$$

Figure 19: Randomly generated queries.