Practical Relational Calculus Query Evaluation

Martin Raszyk David Basin Srđan Krstić
Department of Computer Science
ETH Zürich
Zurich, Switzerland

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

ABSTRACT

The relational calculus is a concise, declarative query language, which allows its users to focus on *what they want* and not on *how to compute it*. However, existing relational calculus 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 relational calculus 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 relational calculus queries. We show that our translation improves the data complexity over existing approaches, which we also empirically confirm in both realistic and synthetic experiments.

CCS CONCEPTS

• **Theory of computation → Database query languages (principles)**.

KEYWORDS

Relational calculus, relative safety, safe-range, query translation

1 INTRODUCTION

Codd's theorem states that all domain-independent queries of the relational calculus (RC) can be expressed in relational algebra (RA) [11]. A popular interpretation of this result is that RA is sufficient to express all interesting queries. This interpretation justifies why SQL evolved as the practical database query language with the RA as its mathematical foundation. However, RC is arguably a more attractive query language than RA as RC is *declarative*: queries can focus what the result should be rather than how to compute it. Being declarative is a burden on the query evaluation engine, since it is not immediately clear how to evaluate queries, but it liberates the query author by making many queries significantly easier to express.

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

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

This query is domain-independent and follows closely the informal description. It is not easy to evaluate, though, in particular because the second conjunct is not domain-independent as it is satisfied for any brand that does not occur in the relation P.

Finding suspicious brands using RA is a nice challenge, which only the best students from an undergraduate database course will accomplish. We give away an answer next (where $\triangleright$ is an anti-join):

$$\pi_b\left(\left(\pi_{u,s}(S) \times B\right) \triangleright \pi_{b,u,s}\left(\left(\pi_{u,s}(S) \times P\right) \triangleright S\right)\right) \cup \left(B \triangleright \pi_b(P)\right).$$

The highlighted expressions are called *generators*. They ensure that the left operands of all anti-join operators include the free variables of the corresponding right operands. (Following Codd's theorem, one would use the active domain for each variable as the generator.)

Van Gelder and Topor [23, 24] present a translation from a decidable class of domain-independent RC queries, called *evaluable*, to RA expressions. In our example, their translation would have different generators, replacing both highlighted parts by $\pi_u(S) \times \pi_s(S)$. The fact that one can avoid this Cartesian product as shown above is subtle. For example, replacing only one of the highlighted generators with the product yields an inequivalent RA expression.

For 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 [3, 6, 14] or to abandon finite relations entirely and evaluate queries using finite representations of well-behaved infinite relations such as systems of constraints [22] or automatic structures [7]. These approaches favor expressiveness over efficiency. Unlike query translations, they cannot benefit from decades of practical research and engineering on databases.

In this work, we translate arbitrary RC queries to RA expressions under the assumption of an infinite domain. To deal with domain-dependent queries, 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 whether the query evaluates to a finite relation for a given database, which can be the case even for domain-dependent queries. 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 allow us to compute the original RC query's finite result, or learn that it would be infinite using RA operations on the unmodified database.

Our translation proceeds in several steps via safe-range queries and relational algebra normal form (Section 3). We focus on the first step of translating RC to two safe-range queries (Section 4), which fundamentally differs from Van Gelder and Topor's approach and produces the better generator $\pi_{u,s}(S)$. This results in a query with strictly improved data complexity (Section 4.4).

The remaining translation steps are more standard. We translate the resulting RA expressions into SQL using the `radb` tool [26]. Along the way to SQL, we leverage different ideas from the literature

to optimize the overall result (Section 5). For example, generalizing Claußen et al. [10]'s idea, we avoid evaluating Cartesian products like $\pi_{u,s}(S) \times P$ in the above translation by using count aggregations.

The overall translation allows us to use standard database management systems to evaluate RC queries. We implement our translation and empirically demonstrate that PostgreSQL outperforms Van Gelder and Topor's and other approaches when using our translation (Section 6). For our evaluation, we use the realistic Amazon review dataset [21] and design a synthetic benchmark that (randomly) generates hard database instances for random RC queries.

In summary, we make the following three main contributions:

- We devise a translation of an arbitrary RC query into a pair of RA expressions as described above. Our translation result's data complexity improves over Van Gelder and Topor's [24].
- We implement our translation and extend it to produce SQL. The resulting tool RC2SQL makes RC a viable input language for standard database management systems. We evaluate our tool on synthetic and realistic data and confirm that our translation's improved data complexity carries over into practice.
- To challenge RC2SQL (and its competitors) in our evaluation, we devise the *Data Golf* benchmark that generates hard database instances for randomly generated RC queries.

2 RELATED WORK

We group the related approaches into three categories.

Structure reduction. The classical approach to handling arbitrary RC queries is to evaluate them under a finite structure [16]. The central question here is whether the evaluation produces the same result as defined by the natural semantics, which typically considers infinite domains. Codd's theorem [11] affirmatively answers this question for domain-independent queries, taking the *active domain* as the finite structure. Ailamazyan et al. [3] show that by extending the active domain with just a few additional elements, whose number depends only on the query, one can decide relative safety for arbitrary RC queries. Further extensions are *natural–active collapse* results that combine the structure reduction with a translation-based approach [6, 14]. All these works are inspiring theoretical landmarks. Yet query evaluation based on (extended) active domains remains impractical due to its prohibitively high data complexity.

Translation-based approaches. Another strategy, which we also follow, is to translate a given query into one that can be evaluated efficiently, for example as a sequence of RA operations. Van Gelder and Topor pioneered this approach [23, 24] for RC. A core component of their translation is the choice of generators, which replace the active domain restrictions from structure reduction approaches, thereby improving the data complexity. Subsequently, extensions to scalar and complex function symbols have been studied [13, 17]. All these approaches focus on syntactic classes of RC, for which domain-independence is given, e.g., *evaluable* queries in case of Van Gelder and Topor (Appendix A). Our approach is inspired by Van Gelder and Topor's but generalizes it to handle arbitrary RC queries at the expense of assuming an infinite underlying domain. Also, we further improve the data complexity of Van Gelder and Topor's approach by choosing better generators.

Infinite relations. Constraint databases [22] obviate the need for using finite tables when evaluating RC queries. This yields significant expressiveness gains over RC. Yet the efficiency of the quantifier elimination procedures employed cannot compare with the simple evaluation of a projection operation in RA. Similarly, automatic structures [7] can represent the results of arbitrary RC queries finitely, but struggle with large data quantities. In our evaluation we compare our translation to several modern incarnations of the above approaches, all based on binary decision diagrams [5, 8, 15, 18, 19].

3 PRELIMINARIES

We now introduce the standard RC syntax and semantics, and define the relevant classes of RC queries [2, Section 5.4].

3.1 Relational Calculus

A signature S is a triple (C, R, ι), where C and R are disjoint finite sets of constant and predicate symbols, and the function $\iota : R \to \mathbb{N}$ maps each predicate symbol $r \in R$ to its arity $\iota(r)$. Let $S = (C, R, \iota)$ be a signature and V a countably infinite set of variables disjoint from $C \cup R$, 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, $r \in R$ is a predicate symbol, $t, t_1, \ldots, t_{\iota(r)} \in V \cup C$ are terms, and $x \in V$ is a variable. We write $\exists \vec{v}.\, Q$ as a shorthand 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 the query Q. We use $\approx$ to denote the equality of terms in RC in order to distinguish it from $=$, which denotes the meta-level equality.

We denote by $|Q|$ the number of constructors in a query Q and by $\mathrm{fv}(Q)$ the set of *free variables* in Q. Furthermore, we denote by $\vec{\mathrm{fv}}(Q)$ 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., $\mathrm{fv}(Q) = \emptyset$, is called *closed*. Queries of the form $r(t_1, \ldots, t_{\iota(r)})$ are called *atomic predicates*. Queries of the form $\exists \vec{v}.\, r(t_1, \ldots, t_{\iota(r)})$ are called *quantified predicates*. A Boolean function $\mathrm{ap}(Q)$ is true iff Q is an atomic predicate. We denote by $\Pi_x(Q)$ the query obtained by existentially quantifying a variable x from a query Q if x is free in Q, i.e., $\Pi_x(Q) :=$ if $x \in \mathrm{fv}(Q)$ then $\exists x.\, Q$ else Q. We lift this notation to sets of queries in the standard way. We use $\Pi_x(Q)$ (instead of $\exists x.\, Q$) to avoid introducing bound variables that never occur in Q.

A structure $\mathcal{D}$ over a signature (C, R, ι) consists of a non-empty domain $\mathbb{D}$ and interpretations $c^{\mathcal{D}} \in \mathbb{D}$ and $r^{\mathcal{D}} \subseteq \mathbb{D}^{\iota(r)}$, for each $c \in C$ and $r \in R$. We assume that all the relations $r^{\mathcal{D}}$ are *finite*. Note that this assumption does *not* yield a finite structure (as defined in finite model theory [16]) since the domain $\mathbb{D}$ can still be infinite. A (*variable*) *assignment* is a mapping $v : V \to \mathbb{D}$. We additionally define v on constant symbols $c \in C$, as $v(c) = c^{\mathcal{D}}$. We write $v[x \mapsto d]$ for the assignment that maps x to $d \in \mathbb{D}$ and is otherwise identical to v. We lift this notation to sequences $\vec{x}$ and $\vec{d}$ of the same length.

Given a structure $\mathcal{D}$ and an assignment v, we define the semantics of RC queries as follows:

$(\mathcal{D}, v) \not\models \bot$; $(\mathcal{D}, v) \models \top$;
$(\mathcal{D}, v) \models (x \approx t)$ iff $v(x) = v(t)$;
$(\mathcal{D}, v) \models r(t_1, \ldots, t_{\iota(r)})$ iff $(v(t_1), \ldots, v(t_{\iota(r)})) \in r^{\mathcal{D}}$;
$(\mathcal{D}, v) \models (\neg Q)$ iff $(\mathcal{D}, v) \not\models Q$;
$(\mathcal{D}, v) \models (Q_1 \vee Q_2)$ iff $(\mathcal{D}, v) \models Q_1$ or $(\mathcal{D}, v) \models Q_2$;
$(\mathcal{D}, v) \models (Q_1 \wedge Q_2)$ iff $(\mathcal{D}, v) \models Q_1$ and $(\mathcal{D}, v) \models Q_2$;
$(\mathcal{D}, v) \models (\exists x.\, Q)$ iff $(\mathcal{D}, v[x \mapsto d]) \models Q$, for some $d \in \mathbb{D}$.

We write $v \models Q$ for $(\mathcal{D}, v) \models Q$ if the structure $\mathcal{D}$ is fixed in the given context. For a fixed $\mathcal{D}$, only on the assignments to Q's free variables influence $v \models Q$, i.e., $v \models Q$ is equivalent to $z \models Q$, for any variable assignment z that agrees with v on $\mathrm{fv}(Q)$. For closed queries Q, we write $\models Q$ and say that Q holds, since such queries either hold for all variable assignments or for none of them. We call a finite sequence $\vec{d}$ of domain elements $d_i \in \mathbb{D}$ a *tuple*. Given a query Q and a structure $\mathcal{D}$, we denote the set of satisfying tuples for Q by

$$[\![Q]\!]^{\mathcal{D}} = \{\vec{d} \in \mathbb{D}^{|\vec{\mathrm{fv}}(Q)|} \mid \exists v.\, (\mathcal{D}, v[\vec{\mathrm{fv}}(Q) \mapsto \vec{d}]) \models Q\}\,.$$

We omit $\mathcal{D}$ from $[\![Q]\!]^{\mathcal{D}}$ if $\mathcal{D}$ is fixed in the given context, and we call values from $[\![Q]\!]$ assigned to $x \in \mathrm{fv}(Q)$ as Q's column x.

Let a structure $\mathcal{D}$ over a signature (C, R, ι) be fixed. The *active domain* $\mathrm{adom}(Q)$ of a query Q is a subset of the domain $\mathbb{D}$ containing the interpretations $c^{\mathcal{D}}$ of all constant symbols that occur in Q and the values in the relations $r^{\mathcal{D}}$ interpreting all predicate symbols that occur in Q. Since C and R are finite and all $r^{\mathcal{D}}$ are finite and have finite arity $\iota(r)$, the active domain $\mathrm{adom}(Q)$ is also finite.

Queries Q and Q' over the same signature are *equivalent*, written $Q \equiv Q'$, if $(\mathcal{D}, v) \models Q \iff (\mathcal{D}, v) \models Q'$, for any $\mathcal{D}$ and v. Queries Q and Q' over the same signature are *inf-equivalent*, written $Q \overset{\infty}{\equiv} Q'$, if $(\mathcal{D}, v) \models Q \iff (\mathcal{D}, v) \models Q'$, for any $\mathcal{D}$ with an *infinite* domain $\mathbb{D}$ and any v. Clearly, equivalent queries are also inf-equivalent.

A query Q is *domain-independent* if $[\![Q]\!]^{\mathcal{D}} = [\![Q]\!]^{\mathcal{D}'}$ holds for any two structures $\mathcal{D}$ and $\mathcal{D}'$ that agree on the interpretations of constants ($c^{\mathcal{D}} = c^{\mathcal{D}'}$) and predicates ($r^{\mathcal{D}} = r^{\mathcal{D}'}$), while their underlying domains $\mathbb{D}$ and $\mathbb{D}'$ may differ. Agreement on the interpretations implies $\mathrm{adom}(Q) \subseteq \mathbb{D} \cap \mathbb{D}'$. It is undecidable whether an RC query is domain-independent [12, 25]. Note that $\top$ is domain-independent, while $P(x) \vee \neg P(x)$ is not, as $[\![P(x) \vee \neg P(x)]\!] = \mathbb{D}$. Still, any two equivalent queries with the *same* free variables are either both domain-independent or neither is domain-independent.

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

We define a function $\mathrm{flat}^{\oplus}(F)$, where $\oplus \in \{\vee, \wedge\}$, that computes a set of queries by "flattening" the operator $\oplus$ in F. Formally:

$$\mathrm{flat}^{\oplus}(F) := \text{if } F = F_1 \oplus F_2 \text{ then } \mathrm{flat}^{\oplus}(F_1) \cup \mathrm{flat}^{\oplus}(F_2) \text{ else } \{F\}$$

3.2 Safe-Range Queries

The class of *safe-range* queries [2] is a decidable subset of domain-independent RC queries. Its definition is based on the notion of range-restricted variables of a query. A variable is called *range-restricted* if "its possible values all lie within the active domain of the query" [2]. Intuitively, atomic predicates restrict the possible values of a variable that occurs in them as a term. A similar idea applies to equalities between a variable and a constant. An equality $x \approx y$ can extend the set of range-restricted variables in a conjunction $A \wedge x \approx y$: If x or y is a range-restricted variable in A, then both x and y are range-restricted variables in $A \wedge x \approx y$.

$$\begin{array}{ll}
\mathrm{gen}(x, \bot, \emptyset);\ \mathrm{gen}(x, x \approx c, \{x \approx c\}); & \\
\mathrm{gen}(x, A, \{A\}) & \text{if } \mathrm{ap}(A) \wedge x \in \mathrm{fv}(A); \\
\mathrm{gen}(x, \neg\neg A, C) & \text{if } \mathrm{gen}(x, A, C); \\
\mathrm{gen}(x, \neg(A \vee B), C) & \text{if } \mathrm{gen}(x, (\neg A) \wedge (\neg B), C); \\
\mathrm{gen}(x, \neg(A \wedge B), C) & \text{if } \mathrm{gen}(x, (\neg A) \vee (\neg B), C); \\
\mathrm{gen}(x, A \vee B, C_1 \cup C_2) & \text{if } \mathrm{gen}(x, A, C_1) \wedge \mathrm{gen}(x, B, C_2); \\
\mathrm{gen}(x, A \wedge B, C) & \text{if } \mathrm{gen}(x, A, C) \vee \mathrm{gen}(x, B, C); \\
\mathrm{gen}(x, A \wedge x \approx y, C[y \mapsto x]) & \text{if } \mathrm{gen}(y, A, C); \\
\mathrm{gen}(x, A \wedge y \approx x, C[y \mapsto x]) & \text{if } \mathrm{gen}(y, A, C); \\
\mathrm{gen}(x, \exists y.\, A, \Pi_y(C)) & \text{if } x \neq y \wedge \mathrm{gen}(x, A, C).
\end{array}$$

Figure 1: The *generated* relation.

We formalize range-restricted variables using the *generated* relation $\mathrm{gen}(x, Q, C)$, defined in Figure 1. Specifically, $\mathrm{gen}(x, Q, C)$ holds if x is a range-restricted variable in Q and any satisfying assignment for Q satisfies some quantified predicate or equality $x \approx c$, referred to as *generator*, from C. Note that, unlike in the similar definition by Van Gelder and Topor [24, Fig. 5], the rule $\mathrm{gen}(x, \exists y.\, A, \Pi_y(C))$ existentially quantifies the bound variable y in all queries in C where y occurs. Thus, $\mathrm{fv}(C) \subseteq \mathrm{fv}(Q)$ holds for any x and Q whenever $\mathrm{gen}(x, Q, C)$ holds. We formalize these relationships as follows.

Lemma 3.1. *Let Q be a query, $x \in \mathrm{fv}(Q)$, and C be a set of quantified predicates and equalities $x \approx c$ such that $\mathrm{gen}(x, Q, C)$. Then (i) $x \in \mathrm{fv}(F) \subseteq \mathrm{fv}(Q)$, for every $F \in C$, (ii) $v \models Q \Longrightarrow v \models \bigvee_{F \in C} F$, for any assignment v, and (iii) $Q[x/\bot]$ is syntactically equal to $\bot$.*

Definition 3.2. Let $\mathrm{gen}(x, Q)$ be defined as $\exists C.\, \mathrm{gen}(x, Q, C)$. A query Q is *safe-range* if $\mathrm{gen}(x, Q)$ holds for every free variable $x \in \mathrm{fv}(Q)$ and $\mathrm{gen}(y, F)$ holds for the bound variable y in every subquery $\exists y.\, F$ of Q. Let $\bar{G}(Q) := \{x \in \mathrm{fv}(Q) \mid \neg\mathrm{gen}(x, Q)\}$ be the set of free variables in a query Q that are not range-restricted. A query Q has *range-restricted bound variables* if the bound variable y in every subquery $\exists y.\, F$ of Q is range-restricted, i.e., $\mathrm{gen}(y, F)$ holds.

Relational algebra normal form (RANF) [2] is a class of safe-range RC queries, which can be easily mapped to RA. In particular, RANF permits the "construction of an equivalent RA query using an induction from leaf to root" [2]. In Appendix C we formally define the $\mathrm{RANF}(Q)$ predicate and the function $\mathrm{SR2RANF}(\cdot)$ that given a safe-range query Q computes an equivalent RANF query $\mathrm{SR2RANF}(Q)$.

3.3 Query Cost

To assess the data complexity of evaluating a RANF query Q, we define the *cost* of Q, denoted $\mathrm{cost}(Q)$, to be the sum of intermediate result sizes over all RANF subqueries of Q. Formally, $\mathrm{cost}(Q) := \sum\{|\,[\![Q']\!]\,| \cdot |\mathrm{fv}(Q')| \mid Q' \sqsubseteq Q,\ \mathrm{RANF}(Q')\}$, where $Q' \sqsubseteq Q$ holds for all subqueries Q' of Q. This is justified as the cases in the definition of a RANF query (Appendix C, Figure 9) correspond to RA operations projection, column duplication, selection, set union, binary join, and anti-join and their complexity is linear in the combined input and output size. The output size (the number of tuples times the number of variables) is counted in the term $|\,[\![Q']\!]\,| \cdot |\mathrm{fv}(Q')|$ and the input size is counted as the output size for the corresponding 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

We now describe our approach to evaluating an arbitrary RC query Q over a fixed structure $\mathcal{D}$ with an infinite domain $\mathbb{D}$. We first translate Q into an inf-equivalent query Q' with range-restricted bound variables. To this end, we combine the approaches of Hull and Su [14] and of Van Gelder and Topor [24]. We also use the ideas from the proof of the natural-active collapse for RC [6, Theorem 2]. Next, we translate Q' into a pair of safe-range queries $(Q_{\text{fin}}, Q_{\text{inf}})$ where:

(1) $\text{fv}(Q_{\text{fin}}) = \text{fv}(Q)$ unless Q_{fin} is syntactically equal to $\bot$,
(2) $\text{fv}(Q_{\text{inf}}) = \emptyset$, i.e., the query Q_{inf} is closed,
(3) $[\![Q]\!] = [\![Q_{\text{fin}}]\!]$ is a finite set if Q_{inf} does not hold, and
(4) $[\![Q]\!]$ is an infinite set if Q_{inf} holds.

Since the queries Q_{fin} and Q_{inf} are safe-range, they are also domain-independent and thus $[\![Q_{\text{fin}}]\!]$ is a finite set of tuples. In particular, $[\![Q]\!]$ is a finite set of tuples if Q_{inf} does not hold.

4.1 Restricting One Variable

Let x be a free variable in a query Q with range-restricted bound variables. This assumption on Q will be discharged by translating the query Q bottom-up (Section 4.2). In this section, we develop a translation of the query Q into an equivalent query Q' such that the variable x is range-restricted except in a single subquery of Q' that is the negation of a disjunction of quantified predicates with a free occurrence of x and equalities of the form $x \approx c$ or $x \approx y$. From the proof of natural-active collapse for RC [6, Theorem 2], we derive the following translation of Q into an equivalent query:

$$Q \equiv (Q \wedge x \in \text{adom}(Q)) \vee \bigvee_{y \in \text{fv}(Q) \setminus \{x\}} (Q[x \mapsto y] \wedge x \approx y) \vee \left(Q[x/\bot] \wedge \neg \left(x \in \text{adom}(Q) \vee \bigvee_{y \in \text{fv}(Q) \setminus \{x\}} x \approx y \right) \right).$$

Here, $x \in \text{adom}(Q)$ stands for an RC query expressing that the single free variable x is in $\text{adom}(Q)$. The translation distinguishes between the following three cases for a fixed valuation v:

- if $x \in \text{adom}(Q)$ holds, then we do not alter the query Q;
- if $x \approx y$ holds for some free variable $y \in \text{fv}(Q) \setminus \{x\}$, then x can be replaced by y in the query Q;
- otherwise, the query Q is equivalent to $Q[x/\bot]$, i.e., all atomic predicates with a free occurrence of x can be replaced by $\bot$ (because $\neg x \in \text{adom}(Q)$), all equalities $x \approx y$ and $y \approx x$ for $y \in \text{fv}(Q) \setminus \{x\}$ can be replaced by $\bot$ (because $x \neq y$ for all $y \in \text{fv}(Q) \setminus \{x\}$), and all equalities $x \approx z$ for a bound variable z can be replaced by $\bot$ (because $\neg x \in \text{adom}(Q)$ and the bound variable z occurs in a subquery $\exists z.\, F$ such that by assumption $\text{gen}(z, F)$ and thus $z \in \text{adom}(F) \subseteq \text{adom}(Q)$).

Let $\exists * x.\, Q$ be the query obtained from Q by existentially quantifying all free variables in Q except x, i.e., $\exists * x.\, Q \coloneqq \exists \vec{\text{fv}}(Q) \setminus \{x\}.\, Q$. Given a set of queries C, we write $\exists * x.\, C$ for $\bigvee_{F \in C} \exists * x.\, F$. To avoid enumerating the entire active domain $\text{adom}(Q)$ of the query Q, Van Gelder and Topor [24] replace the condition $x \in \text{adom}(Q)$ in their translation by $\exists * x.\, C$, where generator set C is a subset of atomic predicates. Because their translation [24] must yield an equivalent query (for any finite or infinite domain), C must satisfy

$$\neg \exists * x.\, C \Longrightarrow Q \equiv Q[x/\bot] \text{ and} \tag{VGT1}$$

$$Q[x/\bot] \Longrightarrow Q\,. \tag{VGT2}$$

$$\begin{aligned}
&\text{cov}(x, A, \emptyset) && \text{if } x \notin \text{fv}(A);\\
&\text{cov}(x, x \approx c, \{x \approx c\}); \text{cov}(x, x \approx x, \emptyset);\\
&\text{cov}(x, x \approx y, \{x \approx y\}) && \text{if } x \neq y;\\
&\text{cov}(x, y \approx x, \{x \approx y\}) && \text{if } x \neq y;\\
&\text{cov}(x, A, \{A\}) && \text{if } \text{ap}(A) \wedge x \in \text{fv}(A);\\
&\text{cov}(x, \neg A, C) && \text{if } \text{cov}(x, A, C);\\
&\text{cov}(x, A \vee B, C_1 \cup C_2) && \text{if } \text{cov}(x, A, C_1) \wedge \text{cov}(x, B, C_2);\\
&\text{cov}(x, A \vee B, C) && \text{if } \text{cov}(x, A, C) \wedge A[x/\bot] = \top;\\
&\text{cov}(x, A \vee B, C) && \text{if } \text{cov}(x, B, C) \wedge B[x/\bot] = \top;\\
&\text{cov}(x, A \wedge B, C_1 \cup C_2) && \text{if } \text{cov}(x, A, C_1) \wedge \text{cov}(x, B, C_2);\\
&\text{cov}(x, A \wedge B, C) && \text{if } \text{cov}(x, A, C) \wedge A[x/\bot] = \bot;\\
&\text{cov}(x, A \wedge B, C) && \text{if } \text{cov}(x, B, C) \wedge B[x/\bot] = \bot;\\
&\text{cov}(x, \exists y.\, A, \Pi_y(C)) && \text{if } x \neq y \wedge \text{cov}(x, A, C) \wedge (x \approx y) \notin C;\\
&\text{cov}(x, \exists y.\, A, \Pi_y(C \setminus \{x \approx y\}) \cup C'[y \mapsto x]) && \\
& && \text{if } x \neq y \wedge \text{cov}(x, A, C) \wedge \text{gen}(y, A, C').
\end{aligned}$$

Figure 2: The *covered* relation.

Note that $Q[x/\bot] \Longrightarrow \forall x.\, Q$ does not hold for the query $Q \coloneqq \neg P(x)$ and thus a generator set C of atomic predicates satisfying (VGT2) only exists for a proper subset of all RC queries. In contrast, we only require that C satisfies (VGT1) in our translation. To this end, we define a *covered* relation $\text{cov}(x, Q, C)$ (in contrast to Van Gelder and Topor's *constrained* relation $\text{con}(x, Q, C)$ defined in Appendix A, Figure 7) such that, for any variable x and query Q with range-restricted bound variables, there exists at least one set C such that $\text{cov}(x, Q, C)$ and (VGT1) holds. Figure 2 shows the definition of this relation. Unlike the generator set C in $\text{gen}(x, Q, C)$, the *cover* set C in $\text{cov}(x, Q, C)$ may also contain equalities between two variables. Hence, we define a function $\mathcal{G}(C)$ that collects all *generators*, i.e., quantified predicates and equalities $x \approx c$ and a function $\mathcal{E}_x(C)$ that collects all *equality variables* y distinct from x occurring in equalities of the form $x \approx y$. We use $\mathcal{G}^\vee(C)$ to denote the query $\bigvee_{P \in \mathcal{G}(C)} P$. We state the soundness and completeness of the relation $\text{cov}(x, Q, C)$ in the following lemma.

Lemma 4.1. *Let Q be a query with range-restricted bound variables and $x \in \text{fv}(Q)$. Then there exists a set C of quantified predicates and equalities such that* $\text{cov}(x, Q, C)$ *and*

$$\neg \left(\mathcal{G}^\vee(C) \vee \bigvee_{y \in \mathcal{E}_x(C)} x \approx y \right) \Longrightarrow Q \equiv Q[x/\bot]\,.$$

Finally, to preserve the dependencies between the variable x and the remaining free variables of Q occurring in the quantified predicates from $\mathcal{G}(C)$, we do not project $\mathcal{G}(C)$ on the single variable x, i.e., we restrict x by $\mathcal{G}^\vee(C)$ instead of $\exists * x.\, \mathcal{G}(C)$. This yields our optimized translation characterized by the following lemma.

Lemma 4.2. *Let Q be a query with range-restricted bound variables, $x \in \text{fv}(Q)$, and C be a set of quantified predicates and equalities such that* $\text{cov}(x, Q, C)$. *Then $x \in \text{fv}(F) \subseteq \text{fv}(Q)$, for every $F \in \mathcal{G}(C)$, and*

$$Q \equiv (Q \wedge \mathcal{G}^\vee(C)) \vee \bigvee_{y \in \mathcal{E}_x(C)} (Q[x \mapsto y] \wedge x \approx y) \vee \left(Q[x/\bot] \wedge \neg \left(\mathcal{G}^\vee(C) \vee \bigvee_{y \in \mathcal{E}_x(C)} x \approx y \right) \right). \tag{★}$$

The free variable x is not guaranteed to be range-restricted in the last disjunct of our translation (★). However, it occurs only in the

negation of a disjunction of quantified predicates with a free occurrence of x and equalities of the form $x \approx c$ or $x \approx y$. 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 (VGT2) holds.

Example 4.3. Consider $Q := \neg\exists z.\, P(y,z) \wedge \neg R(x,z)$ in which we want to restrict the free variable x. Then $\mathrm{cov}(x,Q,C)$ holds for $C = \{\exists z.\, R(x,z)\} = \mathcal{G}(C)$. Hence, the translation (★) yields

$$Q \equiv (Q \wedge (\exists z.\, R(x,z))) \vee ((\neg\exists z.\, P(y,z)) \wedge (\neg\exists z.\, R(x,z)))\,.$$

Note that x is not range-restricted in the second disjunct above. Since $Q \Longrightarrow Q[x/\bot]$ then $Q \equiv (Q \wedge (\exists z.\, R(x,z))) \vee (\neg\exists z.\, P(y,z))$.

4.2 Restricting Bound Variables

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

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

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

$$\exists x.\, \left(Q[x/\bot] \wedge \neg\left(\mathcal{G}^{\vee}(C) \vee \bigvee_{y \in \mathcal{E}_x(C)} x \approx y\right)\right) \overset{\infty}{\equiv} Q[x/\bot]$$

because x does not occur free in $Q[x/\bot]$ and there exists a value for x in the infinite domain $\mathbb{D}$ such that $x \neq y$ holds for all finitely many $y \in \mathcal{E}_x(C)$ and x does not appear among the finitely many values interpreting the quantified predicates and equalities $x \approx c$ in $\mathcal{G}(C)$. This translation is captured by the following lemma.

Lemma 4.4. *Let Q be a query with range-restricted bound variables, $x \in \mathrm{fv}(Q)$, and C be a set of atomic predicates and equalities such that $\mathrm{cov}(x,Q,C)$ holds. Then*

$$\exists x.\, Q \overset{\infty}{\equiv} (\exists x.\, Q \wedge \mathcal{G}^{\vee}(C)) \vee \bigvee_{y \in \mathcal{E}_x(C)} (Q[x \mapsto y]) \vee Q[x/\bot]\,. \quad (★\exists)$$

Our approach for restricting all bound variables recursively applies Lemma 4.4. To do so, we define the recursive function $\mathrm{RB}(Q)$ in Figure 3, 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 $\exists\vec{z}.\, F$, where blocks of consecutive existential quantifiers are processed together. First, $\mathrm{RB}(F)$ is recursively applied on Line 7 to establish the precondition of Lemma 4.4 that the translated query has range-restricted bound variables. Then, we repeatedly apply the translation (★∃) while representing the intermediate result by a set $\mathcal{F}$ of disjuncts. Because existential quantification distributes over disjunction, the individual disjuncts can be processed independently. For every F' added to $\mathcal{F}$ after applying the translation (★∃) to F the set of not range-restricted variables decreases, i.e., $\bar{G}(F')$ is a proper subset of $\bar{G}(F)$. This fact entails the termination of the loop on Lines 8–12. We denote the non-deterministic choice of an object X from a non-empty set $\mathcal{X}$ as $X \leftarrow \mathcal{X}$.

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

```
1  function RB(Q) =
2      switch Q do
3          case ¬F do return ¬RB(F);
4          case F1 ∨ F2 do return RB(F1) ∨ RB(F2);
5          case F1 ∧ F2 do return RB(F1) ∧ RB(F2);
6          case ∃z⃗. F do
7              F := {RB(F)};
8              while ∃F ∈ F. Ḡ(F) ∩ z⃗ ≠ ∅ do
9                  F ← {F ∈ F | Ḡ(F) ∩ z⃗ ≠ ∅};
10                 y ← Ḡ(F) ∩ z⃗;
11                 C ← {C | cov(y, F, C)};
12                 F := (F \ {F}) ∪ {F ∧ G^∨(C)} ∪
                       ⋃_{x∈E_y(C)} {F[y ↦ x]} ∪ {F[y/⊥]};
13             return ⋁_{F∈F} ∃z⃗ ∩ fv(F). F;
14         otherwise do return Q;
```

Figure 3: Range-restricting bound variables.

Example 4.5. Recall the query $Q := \neg\exists z.\, P(y,z) \wedge \neg R(x,z)$ from Example 4.3. After applying the translation (★), the variable x was not range-restricted in the second disjunct $(\neg\exists z.\, P(y,z)) \wedge (\neg\exists z.\, R(x,z))$ because of the subquery $\neg\exists z.\, R(x,z)$. If we consider the query $\exists x.\, Q$, i.e., if the variable x is bound, then we can apply the translation (★∃) yielding

$$\exists x.\, Q \overset{\infty}{\equiv} (\exists x.\, Q \wedge (\exists z.\, R(x,z))) \vee (\neg\exists z.\, P(y,z))\,.$$

Then the bound variable x is restricted in the first disjunct (as before), and does not occur in the second disjunct (because there does exist some value in the infinite domain $\mathbb{D}$ that does not belong to the finite interpretation of the atomic predicate $R(x,z)$ making the subquery $\exists x.\, \neg\exists z.\, R(x,z)$ trivially satisfied).

4.3 Restricting Free Variables

Given an arbitrary query Q, we translate the inf-equivalent query $\mathrm{RB}(Q)$ with range-restricted bound variables into a pair of safe-range queries $(Q_{\mathrm{fin}}, Q_{\mathrm{inf}})$ such that the main properties of our translation (1)–(4) hold. Our translation is based on the following lemma.

Lemma 4.6. *Let a structure $\mathcal{D}$ with an infinite domain $\mathbb{D}$ be fixed. Let x be a free variable in a query Q with range-restricted bound variables and let $\mathrm{cov}(x,Q,C)$ for a set of quantified predicates and equalities C. If $Q[x/\bot]$ is never satisfied, then*

$$Q \equiv (Q \wedge \mathcal{G}^{\vee}(C)) \vee \bigvee_{y \in \mathcal{E}_x(C)} (Q[x \mapsto y] \wedge x \approx y)\,, \qquad (☆)$$

If $Q[x/\bot]$ is satisfied by some tuple, then $[\![Q]\!]$ is an infinite set.

Proof. If $Q[x/\bot]$ is never satisfied, then the translation (☆) follows from (★). If $Q[x/\bot]$ is satisfied by some tuple t, then the last disjunct in the translation (★) of Q is satisfied by infinitely many tuples t' obtained from t by assigning x any value from the infinite domain $\mathbb{D}$ such that $x \neq y$ holds for all finitely many $y \in \mathcal{E}_x(C)$ and x does not appear among the finitely many values interpreting the quantified predicates and equalities $x \approx c$ from $\mathcal{G}(C)$. □

input: A RC query Q.
output: A pair of safe-range queries $(Q_{\text{fin}}, Q_{\text{inf}})$ such that (1), (2), (3), and (4) from Section 4 hold.

1 **function** split(Q) =
2 $\quad \mathcal{Q}_{\text{fin}} := \{(\text{RB}(Q), \top)\}, \mathcal{Q}_{\text{inf}} := \emptyset$;
3 $\quad$ **while** $\exists (F, E) \in \mathcal{Q}_{\text{fin}}. \bar{G}(F) \neq \emptyset$ **do**
4 $\quad\quad (F, E) \leftarrow \{(F, E) \in \mathcal{Q}_{\text{fin}} \mid \bar{G}(F) \neq \emptyset\}$;
5 $\quad\quad x \leftarrow \bar{G}(F)$;
6 $\quad\quad C \leftarrow \{C \mid \text{cov}(x, F, C)\}$;
7 $\quad\quad \mathcal{Q}_{\text{fin}} := (\mathcal{Q}_{\text{fin}} \setminus \{(F, E)\}) \cup \{(F \wedge \mathcal{G}^{\vee}(C), E)\} \cup \bigcup_{y \in \mathcal{E}_x(C)} \{(F[x \mapsto y], E \wedge x \approx y)\}$;
8 $\quad\quad \mathcal{Q}_{\text{inf}} := \mathcal{Q}_{\text{inf}} \cup \{F[x/\bot]\}$;
9 $\quad$ **while** $\exists (F, E) \in \mathcal{Q}_{\text{fin}}. \mathcal{W}(F, E) \neq \emptyset \vee \text{fv}(F \wedge E) \neq \text{fv}(Q)$ **do**
10 $\quad\quad (F, E) \leftarrow \{(F, E) \in \mathcal{Q}_{\text{fin}} \mid \mathcal{W}(F, E) \neq \emptyset \vee \text{fv}(F \wedge E) \neq \text{fv}(Q)\}$;
11 $\quad\quad \mathcal{Q}_{\text{fin}} := \mathcal{Q}_{\text{fin}} \setminus \{(F, E)\}$;
12 $\quad\quad \mathcal{Q}_{\text{inf}} := \mathcal{Q}_{\text{inf}} \cup \{F \wedge E\}$;
13 $\quad$ **return** $(\bigvee_{(F,E) \in \mathcal{Q}_{\text{fin}}} (F \wedge E), \text{RB}(\bigvee_{F \in \mathcal{Q}_{\text{inf}}} \exists \vec{\text{fv}}(F).\, F))$;

Figure 4: Restricting free variables by query splitting.

Our approach is implemented by the function split(Q) defined in Figure 4. In the following, we describe this function and informally justify its correctness formalized by the input/output specification. In split(Q), we represent the queries Q_{fin} and Q_{inf} using a set $\mathcal{Q}_{\text{fin}}$ of query pairs and a set $\mathcal{Q}_{\text{inf}}$ of queries such that

$$Q_{\text{fin}} := \bigvee_{(F,E) \in \mathcal{Q}_{\text{fin}}} (F \wedge E)\,, \qquad Q_{\text{inf}} := \bigvee_{F \in \mathcal{Q}_{\text{inf}}} \exists \vec{\text{fv}}(F).\, F\,,$$

and, for any $(F, E) \in \mathcal{Q}_{\text{fin}}$, E is a conjunction of equalities. As long as there exists some $(F, E) \in \mathcal{Q}_{\text{fin}}$ such that $\bar{G}(F) \neq \emptyset$, we apply the translation (☆) to F and add the query $F[x/\bot]$ to $\mathcal{Q}_{\text{inf}}$. We remark that if we applied the translation (☆) to the entire disjunct $F \wedge E$, the loop on Lines 3–8 might not terminate. Note that, for every (F', E') added to $\mathcal{Q}_{\text{fin}}$ after applying the translation (☆) to F, $\bar{G}(F')$ is a proper subset of $\bar{G}(F)$. This entails the termination of the loop on Lines 3–8. Finally, if $[\![F]\!]$ is an infinite set of tuples, then $[\![F \wedge E]\!]$ is an infinite set of tuples, too. This is because the equalities in E merely duplicate columns of the query F. Hence, it indeed suffices to apply the translation (☆) to F instead of the entire disjunct $F \wedge E$.

After the loop on Lines 3–8 in Figure 4 terminates, for any $(F, E) \in \mathcal{Q}_{\text{fin}}$, F is a safe-range query and E is a conjunction of equalities such that $\text{fv}(F \wedge E) = \text{fv}(Q)$. However, the query $F \wedge E$ need not be safe-range, e.g., if $F := P(x)$ and $E := (x \approx y \wedge u \approx v)$. Given a set of equalities E, let $C(E)$ be the set of equivalence classes of free variables $\text{fv}(E)$ with respect to E. For instance, $C(\{a \approx b, b \approx c, d \approx e\}) = \{\{a, b, c\}, \{d, e\}\}$. Let

$$\mathcal{W}(F, E) := \bigcup_{W \in C(\text{flat}^{\wedge}(E)), W \cap \text{fv}(F) = \emptyset} W$$

be the set of all variables in equivalence classes from $C(\text{flat}^{\wedge}(E))$ that are disjoint from F's free variables. Then, $F \wedge E$ is safe-range if and only if $\mathcal{W}(F, E) = \emptyset$ (recall the definition of safe-range).

Now if $\mathcal{W}(F, E) \neq \emptyset$ and $F \wedge E$ is satisfied by some tuple, then $[\![F \wedge E]\!]$ is an infinite set of tuples because all equivalence classes of variables in $\mathcal{W}(F, E) \neq \emptyset$ can be assigned arbitrary values from the infinite domain $\mathbb{D}$. In our example with $F := P(x)$ and $E := (x \approx y \wedge u \approx v)$, we have $\mathcal{W}(F, E) = \{u, v\} \neq \emptyset$. Moreover, if $\text{fv}(F \wedge E) \neq \text{fv}(Q)$ and $F \wedge E$ is satisfied by some tuple, then this tuple can be extended to infinitely many tuples over $\text{fv}(Q)$ by choosing arbitrary values from the infinite domain $\mathbb{D}$ for the variables in the non-empty set $\text{fv}(Q) \setminus \text{fv}(F \wedge E)$. Hence, for any $(F, E) \in \mathcal{Q}_{\text{fin}}$ with $\mathcal{W}(F, E) \neq \emptyset$ or $\text{fv}(F \wedge E) \neq \text{fv}(Q)$, we remove (F, E) from $\mathcal{Q}_{\text{fin}}$ and add $F \wedge E$ to $\mathcal{Q}_{\text{inf}}$. Note that we only remove pairs from $\mathcal{Q}_{\text{fin}}$, which entails the termination of the loop on Lines 9–12. Afterwards, the query Q_{fin} is safe-range. However, the query Q_{inf} need not be safe-range. Indeed, any query $F \in \mathcal{Q}_{\text{inf}}$ has range-restricted bound variables, but not all the free variables of F need be range-restricted and thus the query $\exists \vec{\text{fv}}(F).\, F$ need not be safe-range. But the query Q_{inf} is closed and thus the inf-equivalent query $\text{RB}(Q_{\text{inf}})$ with range-restricted bound variables is safe-range.

Lemma 4.7. *Let Q be an RC query and* $\text{split}(Q) = (Q_{\text{fin}}, Q_{\text{inf}})$. *Then the queries Q_{fin} and Q_{inf} are safe-range;* $\text{fv}(Q_{\text{fin}}) = \text{fv}(Q)$ *unless Q_{fin} is syntactically equal to $\bot$; and* $\text{fv}(Q_{\text{inf}}) = \emptyset$.

Lemma 4.8. *Let a structure $\mathcal{D}$ with an infinite domain $\mathbb{D}$ be fixed. Let Q be an RC query and* $\text{split}(Q) = (Q_{\text{fin}}, Q_{\text{inf}})$. *If $\not\models Q_{\text{inf}}$, then $[\![Q]\!] = [\![Q_{\text{fin}}]\!]$ is a finite set. If $\models Q_{\text{fin}}$, then $[\![Q]\!]$ is an infinite set.*

Example 4.9. Consider the query $Q := \neg\exists y.\, (R(100, y) \wedge \neg S(x, y))$ by Van Gelder and Topor [24, Example 5.4]. Because Q has range-restricted bound variables, we can apply split(Q) obtaining

$$\text{split}(Q) := (Q \wedge (\exists y.\, S(x, y)), \neg\exists y.\, R(100, y))\,.$$

Applying the translation (★) to Q for the free variable x yields

$$Q \equiv (Q \wedge (\exists y.\, S(x, y))) \vee ((\neg\exists y.\, R(100, y)) \wedge (\neg\exists y.\, S(x, y)))\,.$$

If the subquery $\neg\exists y.\, R(100, y)$ from the second disjunct holds, then Q is satisfied by infinitely many values for x from the infinite domain $\mathbb{D}$ that do not belong to the finite interpretation of the atomic predicate $S(x, y)$ and thus satisfy the subquery $\neg\exists y.\, S(x, y)$ from the second disjunct. Hence, the set $[\![Q]\!]$ is an infinite set of tuples whenever $Q_{\text{inf}} := \neg\exists y.\, R(100, y)$ holds. In contrast, if $\neg\exists y.\, R(100, y)$ does not hold, then the query Q is equivalent to the query $Q_{\text{fin}} := Q \wedge \exists y.\, S(x, y)$ obtained by applying the translation (☆) to Q for the free variable x. Note that both Q_{fin} and Q_{inf} are safe-range.

4.4 Complexity Analysis

For our complexity analysis, we assume that the free and bound variables in any query are pairwise distinct. Let $\text{av}(Q)$ be the set of all free and bound variables in a query Q. We denote by $\mathcal{A}(Q)$ the set of atomic predicates in a query Q. Given an atomic predicate $P \in \mathcal{A}(Q)$, we denote by $\mathcal{S}^{Q}(P)$ the list of bound variables in Q on the path from Q's topmost constructor to the occurrence of P. Note that the list $\mathcal{S}^{Q}(P)$ is the same for all occurrences of P in Q because all free and bound variables in Q are assumed to be pairwise distinct. Furthermore, we denote by $\mathcal{A}^{\exists}(Q)$ the set of quantified predicates obtained by projecting away suffixes of $\mathcal{S}^{Q}(P)$ from $P \in \mathcal{A}(Q)$. We provide a formal definition of $\mathcal{A}^{\exists}(Q)$ in Appendix D, Figure 13.

Since the query cost is defined for RANF queries, given an arbitrary RC query Q, we consider the cost of the pair of RANF queries $\text{rw}(Q) = (\hat{Q}_{\text{fin}}, \hat{Q}_{\text{inf}})$ obtained by applying the function

SR2RANF(·) (Appendix C) to the safe-range queries Q_{fin} and Q_{inf} computed by our translation split(Q). We are interested in the data complexity of evaluating the query Q, so we ignore the complexity of the translation itself and focus on the complexity of evaluating the queries $\hat{Q}_{\text{fin}}$ and $\hat{Q}_{\text{inf}}$, i.e., we estimate cost($\hat{Q}_{\text{fin}}$) and cost($\hat{Q}_{\text{inf}}$).

When restricting a variable by $\mathcal{G}^{\vee}(C)$ in our translation, we effectively project away suffixes of $\mathcal{S}^{Q}(P)$ for some $P \in \mathcal{A}(Q)$ and thus we can show that $\mathcal{A}^{\exists}(\hat{Q}_{\text{fin}})$ and $\mathcal{A}^{\exists}(\hat{Q}_{\text{inf}})$ are subsets of $\mathcal{A}^{\exists}(Q)$. We formalize this observation, which is central for our complexity analysis, in the following lemma.

Lemma 4.10. *Let Q be an RC query and let* $\text{rw}(Q) = (\hat{Q}_{\text{fin}}, \hat{Q}_{\text{inf}})$. *Then $\mathcal{A}^{\exists}(\hat{Q}_{\text{fin}}) \subseteq \mathcal{A}^{\exists}(Q)$ and $\mathcal{A}^{\exists}(\hat{Q}_{\text{inf}}) \subseteq \mathcal{A}^{\exists}(Q)$.*

We bound the data complexity of query evaluation in the following theorem. Similarly to Ngo et al. [20], we use the notation $\tilde{O}(\cdot)$ to hide a possible log-factor incurred by set operations.

Theorem 4.11. *Let Q be an RC query, the data complexity of checking if $[\![Q]\!]$ is finite and computing $[\![Q]\!]$ if it is finite is in*

$$\tilde{O}\left(\prod_{P \in \mathcal{A}(Q)} |\,[\![P]\!]\,|\right),$$

i.e., it is proportional to the product of the cardinalities of sets of tuples satisfying the atomic predicates in Q.

We justify and prove Theorem 4.11 in Appendix D. Finally, we show why the data complexity from Theorem 4.11 cannot be achieved by the translation proposed by Van Gelder and Topor [24].

Example 4.12. Consider the query $Q \coloneqq B(b) \wedge \exists u, s.\, \neg\exists p.\, P(b,p) \wedge \neg S(p,u,s)$, equivalent to Q^{susp} from introduction. Then we have $\mathcal{A}^{\exists}(Q) = \{B(b), P(b,p), \exists p.\, P(b,p), S(p,u,s), \exists p.\, S(p,u,s), \exists s, p.\, S(p,u,s), \exists u,s,p.\, S(p,u,s)\}$. The translation Q_{vgt} by Van Gelder and Topor [24] restricts the bound variables r and s by $\exists s, p.\, S(p,u,s)$ and $\exists u, p.\, S(p,u,s)$, respectively. In fact, the latter query is not in $\mathcal{A}^{\exists}(Q)$, which shows that $\mathcal{A}^{\exists}(Q_{\text{vgt}}) \subseteq \mathcal{A}^{\exists}(Q)$ (as well as Lemma 4.10) does not hold for the RANF query Q_{vgt}. Computing the join of $\exists s, p.\, S(p,u,s)$ and $\exists u, p.\, S(p,u,s)$, which is a Cartesian product, yields a data complexity up to $|\,[\![S(p,u,s)]\!]\,|^2$ for Q_{vgt}.

5 IMPLEMENTATION AND OPTIMIZATION

We have implemented our translation RC2SQL consisting of roughly 1000 lines of OCaml code [4]. Although our translation satisfies the worst-case complexity bound stated above, we further improve its average-case complexity. We empirically show that for some queries these optimizations improve evaluation performance by a 10^4 factor.

First, we optimize the nondeterministic choices in our algorithms (Appendix E.1). In short, we use a sample structure of constant size, called a *training database*, to calculate the query cost of different quantified predicates and then pick the smallest one. More specifically, the choices of y and C in Figure 3 are such that we minimize $|\mathcal{E}_y(C)|$ as the primary objective and then minimize $\sum_{P \in \mathcal{G}(C)} \text{cost}(P)$ as the secondary objective.

We use the function OPTCNT(·) (Appendix E.2) implementing an optimization for RANF queries of the form $\exists \vec{y}.\, P \wedge \bigwedge_{i=1}^{n} \neg R_i$ using the count aggregation operator, inspired by the approach of Claußen et al. [10]. Let $\vec{x} \coloneqq \vec{\text{fv}}(P) \setminus \vec{y}$. The key idea is that the above query is satisfied if for each $\vec{x}$ the number of valuations of $\vec{y}$ of P and $\bigvee_{i=1}^{n}(P \wedge R_i)$ is different. Dually, if the above query appears under a negation, the number of valuations of $\vec{y}$ of the two respective queries must be the same.

Finally, the last step in our translation is to generate an SQL query from the (optimized) RANF query by applying the function RANF2SQL(·) (Appendix E.3). We first obtain an equivalent RA expression. We improve upon the standard approach [2] by handling the case of closed RC queries [9] and optimizing the translation of RANF queries of the form $Q \wedge x \approx y$. Specifically, our version of the translation duplicates the attribute x in $[\![Q]\!]$ and renames it to y, instead of performing a natural (self-)join followed by a selection operator. To translate RA expressions into SQL, we reuse a publicly available RA interpreter called `radb` [26]. We modify the implementation of the interpreter to further improve the performance of the resulting SQL query. We map the generalized difference operator Q **diff** Q' to a more efficient LEFT JOIN, when $\text{fv}(Q) \subsetneq \text{fv}(Q')$ and we perform common subquery elimination.

Appendix E provides more details on the implementation of our translation and on the optimizations we employ.

6 EMPIRICAL EVALUATION

To validate our translation's improved data complexity, we compare it with the translation by Van Gelder and Topor [24] (VGT), an implementation of the algorithm by Ailamazyan et al. [3] that uses an extended active domain as the generators, and the DDD [18, 19], LDD [8], and MonPoly-REG [5] tools that support direct RC query evaluation using binary decision diagrams.

The translation VGT defined for evaluable RC queries is obtained from our translation by modifying the function RB(·) in Figure 3 to use the relation $\text{con}(x, Q, C)$ (Appendix A, Figure 7) instead of $\text{cov}(x, Q, C)$ (Figure 2) and to use the generator $\exists * y.\, \mathcal{G}^{\vee}(C)$ instead of $\mathcal{G}^{\vee}(C)$. Evaluable queries Q are always translated into $(Q_{\text{fin}}, \bot)$ by rw(·) because all of Q's free variables are range-restricted. Finally, we use the same functions from our translation to convert the RANF query Q_{fin} obtained from the VGT translation into an SQL query. We also consider translation variants that omit the count aggregation optimization OPTCNT(·), marked with an asterisk (*).

SQL queries computed by both translations are evaluated using the PostgreSQL database engine. We have also used the MySQL database engine in all our experiments but decided to omit it from our evaluation after discovering that it computed incorrect results for some queries. This issue was reported and subsequently confirmed by MySQL developers. We run our experiments on an Intel Core i5-4200U CPU computer with 8 GB RAM. The relations in PostgreSQL are recreated before each invocation to prevent optimizations based on caching recent query evaluation results. We provide all our experiments in an easily reproducible artifact [4].

In our experiments, we use pseudorandom structures generated by our benchmark *Data Golf* (Appendix F), which has two objectives. The first resembles the *regex golf* game's objective [1] (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 $\mathcal{P}$ and $\mathcal{N}$, representing valuations of $\text{fv}(Q)$, Data Golf produces a

Experiment Small, Evaluable pseudorandom queries Q, $|Q| = 14$, $n = 500$:

Translation	0.0	0.0	0.5	0.0	0.1	0.0	0.0	0.0	0.5	0.0
RC2SQL	0.5	**0.2**	**0.3**	**0.2**	**0.4**	**0.2**	1.0	**0.4**	0.8	**0.3**
RC2SQL*	**0.2**	**0.2**	TO	0.3	0.5	**0.2**	**0.2**	**0.4**	**0.5**	**0.3**
VGT	5.6	1.7	**0.3**	2.4	**0.4**	3.2	1.1	**0.4**	0.7	**0.3**
VGT*	12.2	3.0	TO	9.4	23.4	6.0	24.7	5.2	6.7	7.3
DDD	11.5	2.8	RE	5.4	87.1	5.4	5.6	5.7	7.0	RE
LDD	50.3	17.2	186.4	23.0	TO	27.8	58.1	45.8	53.0	TO
MonPoly-REG	59.3	29.7	120.4	37.8	64.3	39.1	65.5	33.4	62.1	197.6

Experiment Medium, Evaluable pseudorandom queries Q, $|Q| = 14$, $n = 20000$:

Translation	0.0	0.0	0.5	0.0	0.1	0.0	0.0	0.0	0.5	0.0
RC2SQL	2.0	1.1	**6.3**	1.1	**1.9**	0.9	TO	1.3	TO	4.5
RC2SQL*	**1.5**	**0.8**	TO	**0.9**	TO	**0.8**	**1.9**	**1.1**	**24.2**	**3.2**
VGT	TO	TO	**6.3**	TO	**1.9**	TO	TO	1.3	TO	6.8
VGT*	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

Experiment Large, Evaluable pseudorandom queries Q, $|Q| = 14$, tool = RC2SQL:

$n = 40000$	2.8	2.3	9.9	2.1	3.1	1.7	TO	2.6	TO	9.2
$n = 80000$	6.3	4.5	15.2	4.2	6.4	3.7	TO	4.9	TO	18.8
$n = 120000$	8.4	6.8	22.1	6.8	9.4	5.4	TO	7.7	TO	27.7

Experiment Inf, Non-evaluable pseudorandom queries Q, $|Q| = 7$, $n = 4000$:

	Infinite results					Finite results				
Translation	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RC2SQL	**0.1**	**0.1**	**0.1**	**0.2**	**0.1**	3.1	**0.3**	0.3	**3.0**	**0.2**
RC2SQL*	**0.1**	**0.1**	**0.1**	**0.2**	**0.1**	**0.3**	**0.3**	**0.2**	TO	0.3
DDD	40.8	43.5	50.3	121.4	83.9	42.7	34.8	32.1	51.8	41.3
LDD	TO	TO	TO	TO	TO	TO	250.7	220.1	TO	266.5
MonPoly-REG	TO	TO	TO	TO	TO	TO	199.2	199.8	270.5	267.2

TO = Timeout of 300s, RE = Runtime Error

Figure 5: Experiments Small, Medium, Large, and Inf.

pseudorandom structure $\mathcal{D}$, such that $\mathcal{P} \subseteq [\![Q]\!]$, $\mathcal{N} \cap [\![Q]\!] = \emptyset$, and $|[\![Q']\!]|$, $|[\![\neg Q']\!]|$ contain at least $\min\{|\mathcal{P}|, |\mathcal{N}|\}$ tuples each, for any $Q' \sqsubseteq Q$. Data Golf assumes that Q has only constrained [24] bound variables, no atomic predicates without free variables, no closed subqueries, and no repeated predicates. We control the size of $\mathcal{D}$ in our experiments using a size parameter $n = |\mathcal{P}| = |\mathcal{N}|$.

In the Small, Medium, and Large experiments, we generate ten pseudorandom queries Q with fixed sizes. The queries satisfy the Data Golf assumptions along with a few additional ones: the queries are not safe-range, have no repeated equalities, disjunction only appears at the top-level, and only pairwise distinct variables appear as terms in predicates. The queries have 14 constructors, 2 free variables, and any subquery has at most 4 free variables. Other parameters in each experiment are the following:

Small: All tools; Structure size $n = 500$
Medium: All tools; Structure size $n = 20000$, and
Large: RC2SQL tool; Structure sizes $n = 40000, 80000, 120000$.

The Inf experiment considers five pseudorandom queries Q that are *not* evaluable, i.e., $\mathrm{rw}(Q) = (Q_{\mathsf{fin}}, Q_{\mathsf{inf}})$, such that $Q_{\mathsf{inf}} \neq \bot$. Specifically, the queries are of the form $\alpha \wedge \forall x, y.\ \beta \longrightarrow \gamma$, where α, β, and γ are either atomic predicates or equalities. For each query Q, we compare the performance of our tool to tools that directly evaluate Q on pseudorandom structures generated by two variants of the Data Golf distribution (parameter β in Appendix F), which trigger infinite or finite evaluation results on the chosen formulas. For infinite results, our tool outputs this fact (by evaluating Q_{inf}), whereas the other tools also output a finite representation of the infinite result. For finite results, all tools produce the same output.

Figure 5 shows the evaluation results for the experiments Small, Medium, Large, and Inf. Our translation used a Data Golf structure with $n = 4$ as its training database. All entries are execution times in seconds, TO is a timeout, and RE is a runtime error. RC2SQL translation's time is shown in the first row of each table. Each column shows times for a unique pseudorandom query. The lowest time for a query is typeset in bold. We omit the rows for tools that time out or crash on all formulas of an experiment. This particularly affects Ailamazyan et al. [3], which times out in all experiments.

	Query Q^{susp}				Query $Q^{\mathrm{susp}\prime}$			
	$m = 10^3$	$m = 10^4$	GC	MI	$m = 10^3$	$m = 10^4$	GC	MI
Translation	0.0	0.0	0.0	0.0	2.5	2.5	0.6	0.6
RC2SQL	0.2	**0.4**	**3.2**	**37.9**	0.3	**4.9**	**8.4**	**317.8**
RC2SQL*	0.9	73.2	496.4	TO	1.2	160.3	TO	TO
VGT	0.2	**0.4**	4.4	40.7	2.7	331.6	TO	TO
VGT*	6.3	TO	TO	TO	TO	TO	TO	TO
DDD	0.2	19.7	97.7	TO	**0.1**	14.4	210.6	TO
LDD	**0.1**	54.9	TO	TO	**0.1**	49.7	TO	TO
MonPoly-REG	6.2	TO	TO	TO	6.4	TO	TO	TO

GC = Gift Cards dataset, MI = Musical Instruments dataset, TO = Timeout of 600s

Figure 6: Experiment with the queries Q^{susp} and $Q^{\mathrm{susp}\prime}$.

We conclude that our translation RC2SQL significantly outperforms all other tools on all queries and scales well to higher values of n, i.e., larger relations in the Data Golf structures, on most queries.

We also evaluate the tools on the query Q^{susp} from introduction and on the query $Q^{\mathrm{susp}\prime} := B(b) \wedge \exists u, s, t.\ \forall p.\ P(b, p) \longrightarrow S(p, u, s) \vee T(p, u, t)$ with an additional relation T that relates user's review text (term t) to a product. We use both pseudorandom and real-world structures obtained from the Amazon review dataset [21]. The pseudorandom relation P fixes m products of roughly $m/8$ brands in total, B contains all brands having at least three products, and S contains pseudorandom reviews of the products with scores $s \in \{1, 2, \ldots, 5\}$ such that roughly one third of the brands from B satisfies the query Q^{susp}. Uniformly distributed pseudorandom relations with $m = 10$ are also used as a training database. The real-world relations P, S, and T are obtained by projecting the respective tables from the Amazon review dataset for some chosen product categories and the relation B contains all brands from the corresponding table that have at least three products. Because the tool by Ailamazyan et al., DDD, LDD, and MonPoly-REG only support integer data, we injectively remap the string and floating-point values from the tables to integers. The training database is obtained by sampling fixed numbers of brands (with at least three products), products (from each sampled brand), reviews (for each sampled product).

The evaluation results are shown in Figure 6. The two parts of the table correspond to the queries Q^{susp} and $Q^{\mathrm{susp}\prime}$ and have the same format: the first two columns show execution times on pseudorandom structures and the last two columns show execution times on tables from two specific product categories from the Amazon review dataset. We conclude that our translation significantly outperforms all other tools on both pseudorandom and real-world structures. In terms of data-complexity, the VGT translation produces a worse safe-range formula than ours when translating Q^{susp}. However, the optimization $\mathrm{OPTCNT}(\cdot)$ manages to rectify this inefficiency and thus VGT exhibits a comparable performance as ours on Q^{susp}. In contrast, the optimization no longer helps for the $Q^{\mathrm{susp}\prime}$ query.

7 CONCLUSION

We presented a translation-based approach to evaluating arbitrary relational calculus queries over an infinite domain with improved data 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.

REFERENCES

[1] 2013. Regex Golf. https://alf.nu/RegexGolf.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases* (1st ed.). Addison-Wesley, USA. http://webdam.inria.fr/Alice/

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

[4] Anonymous Author(s). 2021. Implementation and evaluation associated with this paper. https://github.com/rc2sql/rc2sql.

[5] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. 2015. Monitoring Metric First-Order Temporal Properties. *J. ACM* 62, 2, Article 15 (May 2015), 45 pages. https://doi.org/10.1145/2699444

[6] Michael Benedikt and Leonid Libkin. 2000. Relational Queries over Interpreted Structures. *J. ACM* 47, 4 (July 2000), 644–680. https://doi.org/10.1145/347476.347477

[7] Achim Blumensath and Erich Grädel. 2004. Finite Presentations of Infinite Structures: Automata and Interpretations. *Theory Comput. Syst.* 37, 6 (2004), 641–674. https://doi.org/10.1007/s00224-004-1133-y

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

[9] J. Chomicki and D. Toman. 1995. Implementing temporal integrity constraints using an active DBMS. *IEEE Transactions on Knowledge and Data Engineering* 7, 4 (1995), 566–582. https://doi.org/10.1109/69.404030

[10] Jens Claußen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. 1997. Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases. In *VLDB 1997*, Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld (Eds.). Morgan Kaufmann, 286–295. http://www.vldb.org/conf/1997/P286.PDF

[11] E. F. Codd. 1972. Relational Completeness of Data Base Sublanguages. *Research Report / RJ / IBM / San Jose, California* RJ987 (1972).

[12] Robert A. Di Paola. 1969. The Recursive Unsolvability of the Decision Problem for the Class of Definite Formulas. *J. ACM* 16, 2 (April 1969), 324–327. https://doi.org/10.1145/321510.321524

[13] Martha Escobar-Molano, Richard Hull, and Dean Jacobs. 1993. Safety and Translation of Calculus Queries with Scalar Functions. In *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '93)*. ACM, New York, NY, USA, 253–264. https://doi.org/10.1145/153850.153909

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

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

[16] Leonid Libkin. 2004. *Elements of Finite Model Theory*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-07003-1_1

[17] Hong-Cheu Liu, Jeffrey Xu Yu, and Weifa Liang. 2008. Safety, Domain Independence and Translation of Complex Value Database Queries. *Inf. Sci.* 178, 12 (June 2008), 2507–2533. https://doi.org/10.1016/j.ins.2008.02.005

[18] Jesper B. Møller. 2002. DDDLIB: A Library for Solving Quantified Difference Inequalities. In *CADE-18 (LNCS, Vol. 2392)*, Andrei Voronkov (Ed.). Springer, 129–133. https://doi.org/10.1007/3-540-45620-1_9

[19] Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. 1999. Difference Decision Diagrams. In *CSL 1999 (LNCS, Vol. 1683)*, Jörg Flum and Mario Rodríguez-Artalejo (Eds.). Springer, 111–125. https://doi.org/10.1007/3-540-48168-0_9

[20] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (February 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[21] Jianmo Ni, Jiacheng Li, and Julian J. McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *EMNLP-IJCNLP 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 188–197. https://doi.org/10.18653/v1/D19-1018

[22] Peter Z. Revesz. 2002. *Introduction to Constraint Databases*. Springer. https://doi.org/10.1007/b97430

[23] Allen Van Gelder and Rodney W. Topor. 1987. Safety and Correct Translation of Relational Calculus Formulas. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '87)*. ACM, New York, NY, USA, 313–327. https://doi.org/10.1145/28659.28693

[24] Allen Van Gelder and Rodney W. Topor. 1991. Safety and Translation of Relational Calculus. *ACM Trans. Database Syst.* 16, 2 (May 1991), 235–278. https://doi.org/10.1145/114325.103712

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

[26] Jun Yang. 2019. RA (radb). https://github.com/junyang/radb.

$$
\begin{array}{ll}
\mathrm{gen}^*(x, A, \{A\}) & \text{if } \mathrm{ap}(A) \wedge x \in \mathrm{fv}(A);\\
\mathrm{gen}^*(x, \neg\neg A, C) & \text{if } \mathrm{gen}^*(x, A, C);\\
\mathrm{gen}^*(x, \neg(A \vee B), C) & \text{if } \mathrm{gen}^*(x, (\neg A) \wedge (\neg B), C);\\
\mathrm{gen}^*(x, \neg(A \wedge B), C) & \text{if } \mathrm{gen}^*(x, (\neg A) \vee (\neg B), C);\\
\mathrm{gen}^*(x, \neg\exists y.\, A, C) & \text{if } x \neq y \wedge \mathrm{gen}^*(x, \neg A, C);\\
\mathrm{gen}^*(x, A \vee B, C_1 \cup C_2) & \text{if } \mathrm{gen}^*(x, A, C_1) \wedge \mathrm{gen}^*(x, B, C_2);\\
\mathrm{gen}^*(x, A \wedge B, C) & \text{if } \mathrm{gen}^*(x, A, C);\\
\mathrm{gen}^*(x, A \wedge B, C) & \text{if } \mathrm{gen}^*(x, B, C);\\
\mathrm{gen}^*(x, \exists y.\, A, C) & \text{if } x \neq y \wedge \mathrm{gen}^*(x, A, C);\\
\hline
\mathrm{con}(x, A, \emptyset) & \text{if } x \notin \mathrm{fv}(A);\\
\mathrm{con}(x, A, \{A\}) & \text{if } \mathrm{ap}(A) \wedge x \in \mathrm{fv}(A);\\
\mathrm{con}(x, \neg\neg A, C) & \text{if } \mathrm{con}(x, A, C);\\
\mathrm{con}(x, \neg(A \vee B), C) & \text{if } \mathrm{con}(x, (\neg A) \wedge (\neg B), C);\\
\mathrm{con}(x, \neg(A \wedge B), C) & \text{if } \mathrm{con}(x, (\neg A) \vee (\neg B), C);\\
\mathrm{con}(x, \neg\exists y.\, A, C) & \text{if } x \neq y \wedge \mathrm{con}(x, \neg A, C);\\
\mathrm{con}(x, A \vee B, C_1 \cup C_2) & \text{if } \mathrm{con}(x, A, C_1) \wedge \mathrm{con}(x, B, C_2);\\
\mathrm{con}(x, A \wedge B, C) & \text{if } \mathrm{gen}^*(x, A, C);\\
\mathrm{con}(x, A \wedge B, C) & \text{if } \mathrm{gen}^*(x, B, C);\\
\mathrm{con}(x, A \wedge B, C_1 \cup C_2) & \text{if } \mathrm{con}(x, A, C_1) \wedge \mathrm{con}(x, B, C_2);\\
\mathrm{con}(x, \exists y.\, A, C) & \text{if } x \neq y \wedge \mathrm{con}(x, A, C).
\end{array}
$$

Figure 7: The relations $\mathrm{gen}^*(x, Q, C)$ and $\mathrm{con}(x, Q, C)$ [24].

$$
\begin{array}{ll}
\mathrm{sz}(\bot) = \mathrm{sz}(\top) = \mathrm{sz}(x \approx t) & = 1\\
\mathrm{sz}(r(t_1, \ldots, t_{\iota(r)})) & = 1\\
\mathrm{sz}(\neg A) & = 2 \cdot \mathrm{sz}(A)\\
\mathrm{sz}(A \vee B) & = 2 \cdot \mathrm{sz}(A) + 2 \cdot \mathrm{sz}(B) + 2\\
\mathrm{sz}(A \wedge B) & = \mathrm{sz}(A) + \mathrm{sz}(B) + 1\\
\mathrm{sz}(\exists y.\, A) & = 2 \cdot \mathrm{sz}(A)
\end{array}
$$

Figure 8: The size measure on RC queries.

A EVALUABLE QUERIES

The classes of *evaluable* queries [24, Def. 5.2] and *allowed* queries [24, Def. 5.3] are decidable subsets of domain-independent RC queries. The evaluable queries characterize exactly the domain-independent queries with no repeated predicate symbols [24, Theorem 10.5]. Also, any evaluable query can be translated to an equivalent allowed query [24, Theorem 8.6] and any allowed query can be translated to an equivalent RANF query [24, Theorem 9.6].

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

- every variable $x \in \mathrm{fv}(Q)$ satisfies $\mathrm{gen}^*(x, Q)$, and
- variable y in every subquery $\exists y.\, F$ of Q satisfies $\mathrm{con}(y, F)$.

A query Q is called *allowed* if

- every variable $x \in \mathrm{fv}(Q)$ satisfies $\mathrm{gen}^*(x, Q)$, and
- variable y in every subquery $\exists y.\, F$ of Q satisfies $\mathrm{gen}^*(y, F)$,

where the relations $\mathrm{gen}^*(x, Q)$ and $\mathrm{con}(y, F)$ are $\exists G.\ \mathrm{gen}^*(x, Q, C)$ and $\exists G.\ \mathrm{con}(y, F, C)$, respectively, and the latter ternary homonymous relations are defined in Figure 7.

In Figure 8 we introduce a size measure $\mathrm{sz}(Q)$ on queries, that decreases for proper subqueries after pushing negation and distributing existential quantification over disjunction. We use $\mathrm{sz}(\cdot)$ to show termination of the rules in Figures 1, 2, and 7, as well as the termination of functions in Figures 11 and 12.

We relate the definitions from Figure 1 and Figure 7 with the following lemmas.

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

Proof. The lemma is proved by induction on the query Q using the measure $\mathrm{sz}(Q)$ on queries defined in Figure 8, which decreases in every case of the definition in Figure 7. □

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

Proof. The lemma is proved by induction on the query Q using the measure $\mathrm{sz}(Q)$ on queries defined in Figure 8, which decreases in every case of the definition in Figure 7.

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

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

We remark that there exists a safe-range query that is not allowed, e.g., $P(x) \wedge x \approx y$.

B CONSTANT PROPAGATION

We define the following constant propagation rules:

$$
\begin{array}{lcll}
x \approx x & \equiv & \top, & (C0)\\
\neg\bot & \equiv & \top, & (C1)\\
\neg\top & \equiv & \bot, & (C2)\\
A \wedge \bot & \equiv & \bot, & (C3)\\
\bot \wedge A & \equiv & \bot, & (C4)\\
A \wedge \top & \equiv & A, & (C5)\\
\top \wedge A & \equiv & A, & (C6)\\
A \vee \bot & \equiv & A, & (C7)\\
\bot \vee A & \equiv & A, & (C8)\\
A \vee \top & \equiv & \top, & (C9)\\
\top \vee A & \equiv & \top, & (C10)\\
\exists y.\, \bot & \equiv & \bot, & (C11)\\
\exists y.\, \top & \equiv & \top, & (C12)
\end{array}
$$

Definition B.1. The substitution of the form $Q[x \mapsto y]$ is the query obtained from a query Q by replacing every occurrence of the free variable x with the variable y and performing constant propagation using the rules (C0)–(C12).

Definition B.2. The substitution of the form $Q[x/\bot]$ is the query obtained from a query Q by replacing with $\bot$ every atomic predicate or equality (except for $(x \approx x) \equiv \top$) containing the free variable x and performing constant propagation using the rules (C0)–(C12).

Note that the query $Q[x/\bot]$ is either of the form $\bot$ or $\top$ or contains neither $\bot$ nor $\top$ as a subquery.

$$
\begin{array}{ll}
\mathrm{RANF}(\bot);\ \mathrm{RANF}(\top);\ \mathrm{RANF}(x \approx c); & \\
\mathrm{RANF}(A) & \text{if } \mathrm{ap}(A);\\
\mathrm{RANF}(\neg A) & \text{if } \mathrm{RANF}(A) \wedge \mathrm{fv}(A) = \emptyset;\\
\mathrm{RANF}(A \vee B) & \text{if } \mathrm{RANF}(A) \wedge \mathrm{RANF}(B) \wedge \mathrm{fv}(A) = \mathrm{fv}(B);\\
\mathrm{RANF}(A \wedge B) & \text{if } \mathrm{RANF}(A) \wedge \mathrm{RANF}(B);\\
\mathrm{RANF}(A \wedge \neg B) & \text{if } \mathrm{RANF}(A) \wedge \mathrm{RANF}(B) \wedge \mathrm{fv}(B) \subseteq \mathrm{fv}(A);\\
\mathrm{RANF}(A \wedge (x \approx y)) & \text{if } \mathrm{RANF}(A) \wedge (x \in \mathrm{fv}(A) \vee y \in \mathrm{fv}(A));\\
\mathrm{RANF}(A \wedge \neg(x \approx y)) & \text{if } \mathrm{RANF}(A) \wedge x \in \mathrm{fv}(A) \wedge y \in \mathrm{fv}(A);\\
\mathrm{RANF}(\exists y.\, A) & \text{if } \mathrm{RANF}(A) \wedge y \in \mathrm{fv}(A).
\end{array}
$$

Figure 9: Characterization of RANF queries.

C QUERY NORMAL FORMS

In this section, we introduce relational algebra normal form (RANF), which is a syntactic class of safe-range RC queries that admits a simple construction of an equivalent relational algebra query. We also introduce other normal forms useful for translating safe-range queries to RANF queries. Note that a query normal form concerns the structure of the queries rather than functional dependencies between the attributes in relations (e.g., 1NF, 2NF, 3NF etc.).

Figure 9 defines the Boolean function $\mathrm{RANF}(\cdot)$ characterizing RANF queries. The translation of safe-range queries [2] to equivalent RANF queries can proceed in two different ways based on two different normal forms: safe-range normal form (SRNF) [2] (Appendix C.1), or existential normal form (ENF) [24] (Appendix C.2). A safe-range query in SRNF or ENF can be translated into an equivalent RANF query by subquery rewriting using the following rules [2, Algorithm 5.4.7] [13, Lemma 7.8]:

$$
\begin{array}{lcll}
\psi \wedge (\varphi_1 \vee \varphi_2) & \equiv & (\psi \wedge \varphi_1) \vee (\psi \wedge \varphi_2)\,, & (R1)\\
\psi \wedge (\exists y.\, \varphi) & \equiv & (\exists y.\, \psi \wedge \varphi)\,, & (R2)\\
\psi \wedge \neg\varphi & \equiv & \psi \wedge \neg(\psi \wedge \varphi)\,. & (R3)
\end{array}
$$

Subquery rewriting is a nondeterministic process (as the rewrite rules can be applied in different order) that impacts the performance of evaluating the resulting RANF query. We translate a safe-range query into an equivalent RANF query by a recursive function SR2RANF in a single top-down pass. The function is general in that it also nondeterministically applies rewriting rules, which can be instantiated using different heuristics.

Figure 10 shows an overview of the RC fragments and query normal forms (nodes) and the functions we use to translate between them and to SQL (edges). The dashed edge shows the translation we opt for in this paper. It is the composition of the two translations represented with the solid edges on the left. In the rest of this section we introduce these normal forms and translations. We also justify why we opt for using SRNF to translate safe-range queries to RANF. Appendix E.3 shows how we translate RANF queries to SQL.

C.1 Safe-Range Normal Form

Both the definition of safe-range queries and their translation to equivalent RANF queries is based on the notion of safe-range normal form (SRNF). A query is in SRNF if "the subquery of each negation is either an atomic predicate, equality, or an existentially quantified query" [2]. Figure 11 shows the function $\mathsf{SN}(Q)$ that yields an SRNF query equivalent to a query Q. If the query Q is safe-range, the resulting SRNF query is also safe-range. The function $\mathsf{SN}(Q)$ proceeds by pushing negation [2, Section 5.4], distributing

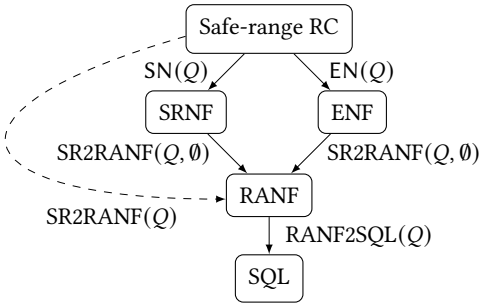

Figure 10: Overview of query normal forms.

input: A RC query Q.
output: An SRNF query Q' such that $Q \equiv Q'$ and $\mathsf{sz}(Q') \leq \mathsf{sz}(Q)$. If Q is safe-range, then Q' is also safe-range.

```
1  function SN(Q) =
2      switch Q do
3          case ¬F do
4              switch F do
5                  case ¬F1 do return SN(F1);
6                  case F1 ∨ F2 do return SN((¬F1) ∧ (¬F2));
7                  case F1 ∧ F2 do return SN((¬F1) ∨ (¬F2));
8                  case ∃y⃗. F' do
9                      if y⃗ ∩ fv(F') = ∅ then return SN(¬F');
10                     else
11                         switch SN(F') do
12                             case F1 ∨ F2 do return
                                SN((¬∃y⃗. F1) ∧ (¬∃y⃗. F2)) ;
13                             otherwise do return
                                ¬∃y⃗ ∩ fv(F'). SN(F');
14             otherwise do return ¬F;
15         case F1 ∨ F2 do return SN(F1) ∨ SN(F2);
16         case F1 ∧ F2 do return SN(F1) ∧ SN(F2);
17         case ∃y⃗. F do
18             switch SN(F) do
19                 case F1 ∨ F2 do return
                    SN((∃y⃗. F1) ∨ (∃y⃗. F2));
20                 otherwise do return ∃y⃗ ∩ fv(F). SN(F);
21         otherwise do return Q;
```

Figure 11: Translation to SRNF.

existential quantifiers over disjunction [24, Rule (T9)], and dropping bound variables that never occur [24, Definition 9.2]. The extra rules proposed by Van Gelder and Topor [24] establish a connection between SRNF and existential normal form (Appendix C.2). The termination of the function $\mathsf{SN}(Q)$ is proved using the measure $\mathsf{sz}(Q)$, defined in Figure 8. We prove the following lemma that we use as a precondition for translating safe-range SRNF to RANF queries.

Lemma C.1. *Let Q be a query in SRNF, then for all Q' such that $(\neg Q') \sqsubseteq Q$, $\mathsf{gen}(x, \neg Q')$ does not hold for any variable x.*

Proof. According to Figure 1, $\mathsf{gen}(x, \neg Q')$ can only hold if $\neg Q'$ has the form $\neg\neg A$, $\neg(A \vee B)$, or $\neg(A \wedge B)$. The SRNF query Q cannot have a subquery $(\neg Q') \sqsubseteq Q$ that is of any of these forms. □

C.2 Existential Normal Form

Here we describe existential normal form (ENF), introduced by Van Gelder and Topor [24] to translate an allowed query [24] into an equivalent RANF query. ENF is an alternative to SRNF in the sense that the same rewrite rules (R1), (R2), and (R3) can be applied to any allowed query in ENF to obtain an equivalent RANF query, using [13, Lemma 7.8]. Unlike SRNF, a query in ENF can have a negation above a conjunction, but not under a disjunction. Moreover, a conjunction of only negations can only appear under an existential quantification. A function $\mathsf{EN}(Q)$ that yields an ENF query equivalent to Q can be defined in terms of subquery rewriting using the rules in [13, Fig. 2]. If the query Q is safe-range, $\mathsf{EN}(Q)$ is also safe-range [13, Lemma 7.4] (note that any safe-range query is embedded-allowed [13]). We prove the following lemma that we use as a precondition for translating safe-range queries in ENF to RANF queries.

Lemma C.2. *Let Q be a query in ENF, then for all Q' such that $(\neg Q') \sqsubseteq Q$, $\mathsf{gen}(x, \neg Q')$ does not hold for any variable x.*

Proof. Assume that $\mathsf{gen}(x, \neg Q')$ holds for a variable x of a subquery $(\neg Q') \sqsubseteq Q$. We derive a contradiction by induction on the number of constructors in Q. According to Figure 1, $\mathsf{gen}(x, \neg Q')$ can only hold if $\neg Q'$ has the form $\neg\neg A$, $\neg(A \vee B)$, or $\neg(A \wedge B)$. The query Q in ENF can only have a subquery $(\neg Q') \sqsubseteq Q$ that is of the form $\neg(A \wedge B)$. Because a conjunction of only negations can only appear under an existential quantification (in particular, not under a negation), the conjunction $A \wedge B$ contains a conjunct C that does not have the form of a negation or conjunction (by flattening $A \wedge B$). Then $\mathsf{gen}(x, \neg(A \wedge B))$ implies $\mathsf{gen}(x, \neg C)$. According to Figure 1, $\mathsf{gen}(x, \neg C)$ can only hold if C is a disjunction. Then $\mathsf{gen}(x, \neg C)$ implies $\mathsf{gen}(x, \neg D)$ for a disjunct D in C that does not have the form of a negation (a query in ENF cannot have a negation directly under a disjunction) or disjunction (by flattening the disjunction C). According to Figure 1, $\mathsf{gen}(x, \neg D)$ can only hold if D is a conjunction. Because a conjunction of only negations can only appear under an existential quantification (in particular, not under a disjunction), D is a subquery of a query in ENF that is a conjunction of not only negations and thus D as well as $\neg D$ are queries in ENF. This allows us to apply the induction hypothesis to the query $\neg D$ in ENF that has strictly less constructors than Q (because D is a proper subquery of Q' and $(\neg Q') \sqsubseteq Q$) and $\mathsf{gen}(x, \neg D)$ holds for $(\neg D) \sqsubseteq (\neg D)$. □

Although applying the rules (R1), (R2), and (R3) to a safe-range query in ENF instead of SRNF may result in a RANF query with fewer constructors, the data complexity, i.e., the cost of the query, can be arbitrarily larger. We illustrate this in the following example.

Example C.3. The safe-range query $Q := P(x, y) \wedge \neg(R(x) \wedge S(y))$ is in ENF and RANF, but not SRNF. Applying the rules (R1), (R2), and (R3) to the equivalent SRNF query yields the RANF query $Q' := (P(x, y) \wedge \neg R(x)) \vee (P(x, y) \wedge \neg S(y))$. The costs of the two queries are $\mathsf{cost}(Q) = 2 \cdot |[\![P(x, y)]\!]| + |[\![R(x)]\!]| + |[\![S(y)]\!]| + 2 \cdot |[\![R(x) \wedge S(y)]\!]| + 2 \cdot |[\![Q]\!]|$ and $\mathsf{cost}(Q') = 4 \cdot |[\![P(x, y)]\!]| + |[\![R(x)]\!]| + |[\![S(y)]\!]| + 2 \cdot |[\![P(x, y) \wedge \neg R(x)]\!]| + 2 \cdot |[\![P(x, y) \wedge \neg S(y)]\!]| + 2 \cdot |[\![Q']\!]|$, respectively. Note that the cost of the former query can be arbitrarily larger if $R(x) \wedge S(y)$ evaluates to a large intermediate result, i.e., $|[\![R(x) \wedge S(y)]\!]| \gg |[\![P(x, y)]\!]|$. In contrast, the cost of the latter query can be only larger by a constant factor, i.e., is asymptotically bounded by the cost of the former query.

The safe-range query $P(x,y) \wedge (R(x) \vee \neg S(y))$ is in SRNF, but not ENF. The equivalent query $P(x,y) \wedge \neg(S(y) \wedge \neg R(x))$ is in ENF, but not SRNF. Applying the rules (R1), (R2), and (R3) yields the RANF queries $(P(x,y) \wedge R(x)) \vee (P(x,y) \wedge \neg S(y))$ and $P(x,y) \wedge \neg(P(x,y) \wedge S(y) \wedge \neg R(x))$, respectively. The costs of these two RANF queries are asymptotically equal.

Note that applying the rule (R1) to an ENF query Q might yield a query that is not in ENF if Q occurs under a negation or an existential quantifier. In that case, the negation can be pushed into the disjunction or the existential quantifier can be distributed over the disjunction to bring the query back into ENF. We formalize these steps as additional rewrite rules:

$$\neg(\varphi_1 \vee \varphi_2) \quad \equiv \quad \neg\varphi_1 \wedge \neg\varphi_2, \qquad (R4)$$
$$\exists x.\, \varphi_1 \vee \varphi_2 \quad \equiv \quad (\exists x.\, \varphi_1) \vee (\exists x.\, \varphi_2). \qquad (R5)$$

Given a query Q, we denote by $\mathcal{RA}(Q)$ the set of equivalent RANF queries obtained by applying the rules (R1), (R2), (R3), (R4), and (R5) to Q. Any safe-range query Q in either SRNF or ENF can be translated to RANF by applying the rules (R1), (R2), and (R3) to Q, using [2, Algorithm 5.4.7] and [13, Lemma 7.8]. Hence, the set $\mathcal{RA}(Q)$ is nonempty whenever Q is a safe-range query in either SRNF or ENF. The rules (R4) and (R5) are not needed to obtain a RANF query although [2, Algorithm 5.4.7] and [13, Lemma 7.8] might still apply them to preserve the query's respective normal form.

Inspired by Example C.3, we conjecture that, given any safe-range query Q, there exists an equivalent RANF query obtained by applying the rules (R1), (R2), (R3), (R4), and (R5) to $\mathsf{SN}(Q)$, whose data complexity (measured by the query cost) is at most a constant factor worse than the data complexity of any equivalent RANF query obtained by applying (R1), (R2), (R3), (R4), and (R5) to $\mathsf{EN}(Q)$, i.e., Q translated to ENF. We confirm this observation empirically and include these findings in our artifact [4]. We thus opt for using SRNF instead of ENF for translating safe-range queries into RANF.

Finally, we remark that the rules (R1), (R2), and (R3) are not sufficient to yield an equivalent RANF query for the original definition of ENF [24]. This issue has been identified and fixed by introducing *embedded allowed* queries [13], a generalization of safe-range queries, and using ENF to translate embedded allowed queries to RANF. Existential normal form has also been used to translate embedded allowed queries to RANF by a follow-up work [17].

C.3 Relational Algebra Normal Form

The function $\mathrm{SR2RANF}(Q,\mathcal{R}) = (Q',\mathcal{R}')$, defined in Figure 12, where SR2RANF stands for *safe-range to relational algebra normal form*, takes a safe-range query $Q \wedge \bigwedge_{R\in\mathcal{R}} R$ and returns a RANF query Q' such that $Q \equiv Q'$ whenever $\bigwedge_{R\in\mathcal{R}} R$ holds. The safe-range query is expressed by a conjunction of a query Q and a set of queries $\mathcal{R}$, where any subset of $\mathcal{R}$ can be used as ψ in the rules (R1), (R2), (R3). The subset $\mathcal{R}' \subseteq \mathcal{R}$ in $\mathrm{SR2RANF}(Q,\mathcal{R}) = (Q',\mathcal{R}')$ encodes the actual $\psi := \bigwedge_{R\in\mathcal{R}'} R$ used in the rules (R1), (R2). Initially, we convert a safe-range query Q into SRNF or ENF and set $\mathcal{R} = \emptyset$. Then both $\mathrm{SR2RANF}(Q) := Q'$ and $\mathrm{SR2RANF}'(Q) := Q''$, where $(Q',\_) := \mathrm{SR2RANF}(\mathsf{SN}(Q),\emptyset)$ and $(Q'',\_) := \mathrm{SR2RANF}(\mathsf{EN}(Q),\emptyset)$ return RANF queries Q' and Q'' equivalent to Q. The termination of the function $\mathrm{SR2RANF}(Q,\mathcal{R})$ is proved using the lexicographic measure $(2\cdot \mathrm{sz}(Q) + 2\cdot\sum_{R\in\mathcal{R}} \mathrm{sz}(R) + 2\cdot|R| + c_Q, \mathrm{sz}(Q))$, where $\mathrm{sz}(Q)$ is defined in Figure 8 and $c_Q = 1$ if Q has the form of an equality between two variables or a negation and $c_Q = 0$ otherwise.

input: A safe-range query $Q \wedge \bigwedge_{R\in\mathcal{R}} R$ such that for all $(\neg Q') \sqsubseteq Q \wedge \bigwedge_{R\in\mathcal{R}} R$, $\mathrm{gen}(x,\neg Q')$ does not hold for any variable x.
output: A RANF query Q' and a subset of queries $\mathcal{R}' \subseteq \mathcal{R}$ such that $\bigwedge_{R\in\mathcal{R}} R \Longrightarrow Q \equiv Q'$, $Q' \Longrightarrow Q \wedge \bigwedge_{R\in\mathcal{R}'} R$, $Q' = \mathrm{cp}(Q')$, and $Q' = \bot \vee \mathrm{fv}(Q) \subseteq \mathrm{fv}(Q') \subseteq \mathrm{fv}(Q) \cup \mathrm{fv}(\mathcal{R})$.

function $\mathrm{SR2RANF}(Q,\mathcal{R}) =$
  if $\mathrm{RANF}(Q)$ **then return** $(Q,\emptyset)$;
  switch Q **do**
    case $x \approx y$ **do return** $\mathrm{SR2RANF}(x \approx y \wedge \bigwedge_{R\in\mathcal{R}} R, \emptyset)$;
    case $\neg F$ **do**
      if F *is safe-range* **then**
        $(F',\_) := \mathrm{SR2RANF}(F,\emptyset)$;
        return $(\neg F',\emptyset)$;
      else return $\mathrm{SR2RANF}(\neg F \wedge \bigwedge_{R\in\mathcal{R}} R, \emptyset)$;
    case $F_1 \vee F_2$ **do**
      $\mathcal{F} := \mathrm{flat}^{\vee}(F_1 \vee F_2)$;
      $\mathcal{R}' \leftarrow \{\mathcal{R}' \subseteq \mathcal{R} \mid \bigvee_{F\in\mathcal{F}}(F \wedge \bigwedge_{R\in\mathcal{R}'} R)$ is safe-range$\}$;
      foreach F **in** $\mathcal{F}$ **do** $(F',\_) := \mathrm{SR2RANF}(F \wedge \bigwedge_{R\in\mathcal{R}'} R, \emptyset)$;
      return $(\mathrm{cp}(\bigvee_{F\in\mathcal{F}} F'), \mathcal{R}')$;
    case $F_1 \wedge F_2$ **do**
      $(\mathcal{P},\mathcal{N}) = \mathrm{split}^{+}(\mathrm{flat}^{\wedge}(F_1 \wedge F_2))$;
      $\mathcal{P}^{=} := \{F \in \mathcal{P} \mid \mathrm{eq}(F)\}$; $\mathcal{P}^{+} := \mathcal{P} \setminus \mathcal{P}^{=}$;
      $\mathcal{N}^{=} := \{\neg F \in \mathcal{N} \mid \mathrm{eq}(F)\}$; $\mathcal{N}^{-} := \mathcal{N} \setminus \mathcal{N}^{=}$;
      foreach F **in** $\mathcal{P}^{+}$ **do** $(F', \mathcal{R}_F) := \mathrm{SR2RANF}(F, \mathcal{R} \cup (\mathcal{P} \setminus \{F\}))$;
      $\overline{\mathcal{P}} \leftarrow \{\overline{\mathcal{P}} \subseteq \mathcal{P}^{+} \mid \mathcal{P}^{+} \subseteq \bigcup_{F\in\overline{\mathcal{P}}}(\mathcal{R}_F \cup \{F\})\}$;
      foreach $\neg F$ **in** $\mathcal{N}^{-}$ **do** $(F',\_) := \mathrm{SR2RANF}(F, \mathcal{R} \cup \mathcal{P})$;
      return $(\mathrm{cp}(\mathrm{sort}^{\wedge}(\bigcup_{F\in\overline{\mathcal{P}}} F' \cup \mathcal{P}^{=} \cup \bigcup_{\neg F\in\mathcal{N}^{-}} \neg F' \cup \mathcal{N}^{=})), \bigcup_{F\in\overline{\mathcal{P}}} \mathcal{R}_F \cap \mathcal{R})$;
    case $\exists \vec{y}.\, F$ **do**
      $\vec{y}' \leftarrow \{\vec{y}' \mid |\vec{y}'| = |\vec{y}| \wedge (\mathrm{fv}(F) \cup \mathrm{fv}(\mathcal{R})) \cap \vec{y}' = \emptyset\}$;
      $F := F[\vec{y} \mapsto \vec{y}']$;
      $\mathcal{R}' \leftarrow \{\mathcal{R}' \subseteq \mathcal{R} \mid F \wedge \bigwedge_{R\in\mathcal{R}'} R$ is safe-range$\}$;
      $(F',\_) := \mathrm{SR2RANF}(F \wedge \bigwedge_{R\in\mathcal{R}'} R, \emptyset)$;
      return $(\mathrm{cp}(\exists \vec{y}'.\, F'), \mathcal{R}')$;
    otherwise do return $(\mathrm{cp}(Q),\emptyset)$;

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

Next we describe the function $\mathrm{SR2RANF}(Q,\mathcal{R})$, defined in Figure 12, which closely follows the approach by Abiteboul et al. [2, Section 5.4]. Note that Abiteboul et al. do not need to perform constant propagation (see Appendix B), i.e., simplifications like $A \wedge \bot \equiv \bot$, $A \vee \bot \equiv A$, etc., because the constants $\bot$ and $\top$ are not in their query syntax. Because $\mathrm{gen}(x,\bot)$ holds and $x \notin \mathrm{fv}(\bot)$, we need to perform constant propagation to guarantee that every disjunct

input: A RC query Q.
output: The set $\mathcal{A}^{\exists}(Q)$ of quantified predicates obtained by projecting away suffixes of $\mathcal{S}^{Q}(P)$ from $P \in \mathcal{A}(Q)$.

function $\mathcal{A}^{\exists}(Q) =$
  switch Q **do**
    case $\exists \vec{v}.\, r(t_1, \ldots, t_{\iota(r)})$ **do return** $\{\mathcal{N}(Q)\}$;
    case $\neg F$ **do return** $\mathcal{A}^{\exists}(F)$;
    case $F_1 \vee F_2$ **do return** $\mathcal{A}^{\exists}(F_1) \cup \mathcal{A}^{\exists}(F_2)$;
    case $F_1 \wedge F_2$ **do return** $\mathcal{A}^{\exists}(F_1) \cup \mathcal{A}^{\exists}(F_2)$;
    case $\exists \vec{v}.\, F$ **do**
      $\mathcal{P}^{\exists} := \mathcal{A}^{\exists}(F)$;
      return $\mathcal{P}^{\exists} \cup \{\mathcal{N}(\exists \vec{v}.\, P^{\exists}) \mid P^{\exists} \in \mathcal{P}^{\exists}\}$;
    otherwise do return $\emptyset$;

Figure 13: The set of quantified predicates.

has the same set of free variables. We flatten the disjunction and conjunction using $\text{flat}^{\vee}(F_1 \vee F_2)$ and $\text{flat}^{\wedge}(F_1 \wedge F_2)$, respectively. In the case of a conjunction $F_1 \wedge F_2$, we additionally split the conjuncts into two sets $\mathcal{P}$ and $\mathcal{N}$ using the function $\text{split}^{+}(\text{flat}^{\wedge}(F_1 \wedge F_2))$ defined as follows. The function $\text{split}^{+}(\mathcal{F}) = (\mathcal{P}, \mathcal{N})$ splits a set of queries into two disjoint sets $\mathcal{P}, \mathcal{N}$ such that $\mathcal{N}$ is the maximal set containing queries whose topmost constructor is negation. Next we split out all equalities between two variables and their negations from the sets $\mathcal{P}$ and $\mathcal{N}$. To this end, we define a Boolean function $\text{eq}(Q)$ on queries characterizing equalities between variables, i.e., $\text{eq}(Q)$ is true if Q has the form $x \approx y$ and false otherwise. Finally, the function $\text{sort}^{\wedge}(\mathcal{F})$ converts a set of queries into a RANF conjunction, defined in Figure 9, i.e., a left-associative conjunction in RANF. Note that the function $\text{sort}^{\wedge}(\mathcal{F})$ must place the queries $x \approx y$ so that either x or y is free in some earlier conjunct, e.g., $P(x) \wedge x \approx y \wedge y \approx z$ is in RANF, but $P(x) \wedge y = z \wedge x \approx y$ is not. In the case of an existentially quantified query $\exists \vec{y}.\, F$, we first rename the variables $\vec{y}$ to fresh variables to avoid capture of the free variables in the set of queries $\mathcal{R}$.

Note that the same query ψ cannot be used twice when applying the rules (R1), (R2) multiple times in a row, e.g., to a query

$$\psi \wedge (\varphi_1 \vee \varphi_2) \wedge (\varphi_3 \vee \varphi_4)$$

that could only be translated into

$$(\psi \wedge \varphi_1 \wedge \varphi_3) \vee (\psi \wedge \varphi_1 \wedge \varphi_4) \vee (\psi \wedge \varphi_2 \wedge \varphi_3) \vee (\psi \wedge \varphi_2 \wedge \varphi_4)\,,$$

but not into

$$((\psi \wedge \varphi_1) \vee (\psi \wedge \varphi_2)) \wedge ((\psi \wedge \varphi_3) \vee (\psi \wedge \varphi_4))\,.$$

Trying to rule out the latter translation would unnecesarily complicate our recursive function $\text{SR2RANF}(Q, \mathcal{R})$. Hence, to preserve ψ, we use the following rules in addition to (R1), (R2), and (R3):

$$\psi \wedge (\varphi_1 \vee \varphi_2) \quad \equiv \quad \psi \wedge ((\psi \wedge \varphi_1) \vee (\psi \wedge \varphi_2))\,, \qquad (R1')$$
$$\psi \wedge (\exists y.\, \varphi) \quad \equiv \quad \psi \wedge (\exists y.\, \psi \wedge \varphi)\,. \qquad (R2')$$

Finally, the nondeterministic choices in the function $\text{SR2RANF}(Q, \mathcal{R})$ are resolved by minimizing the cost of the resulting RANF query with respect to a training database (Appendix E.1).

D COMPLEXITY ANALYSIS

The set $\mathcal{A}^{\exists}(Q)$ of quantified predicates obtained by projecting away suffixes of $\mathcal{S}^{Q}(P)$ from $P \in \mathcal{A}(Q)$ is defined in Figure 13. The quantified predicates in $\mathcal{A}^{\exists}(Q)$ are reduced to a normal form with respect to α-conversion, i.e., renaming of bound variables. We denote by $\mathcal{N}(P^{\exists})$ the quantified predicate obtained by transforming a quantified predicate $P^{\exists}$ into an α-conversion normal form. For instance, $\mathcal{N}(\exists y.\, P(x, y)) = \mathcal{N}(\exists z.\, P(x, z))$.

Next we observe that every tuple satisfying a RANF query $\hat{Q}$ belongs to the set of tuples satisfying the join over a subset $\mathcal{P}^{\exists} \subseteq \mathcal{A}^{\exists}(\hat{Q})$ and equalities $E \in \mathcal{E}$ duplicating some columns from $\mathcal{P}^{\exists}$. To this end, we define $C(W, W')$ to be the set of all equalities $x \approx y$ between variables $x \in W$ and $y \in W'$.

Lemma D.1. *Let $\hat{Q}$ be a RANF query. Then $[\![\hat{Q}]\!]$ satisfies*

$$[\![\hat{Q}]\!] \subseteq \bigcup_{\substack{\mathcal{P}^{\exists} \subseteq \mathcal{A}^{\exists}(\hat{Q}) \\ \mathcal{E} \subseteq C(\text{fv}(\mathcal{P}^{\exists}), \text{fv}(\hat{Q})) \\ \text{fv}(\mathcal{P}^{\exists}) \cup \text{fv}(\mathcal{E}) = \text{fv}(\hat{Q})}} \left[\!\!\left[\bigwedge_{P^{\exists} \in \mathcal{P}^{\exists}} P^{\exists} \wedge \bigwedge_{E \in \mathcal{E}} E \right]\!\!\right].$$

Proof. The statement is proved by well-founded induction over the inductive definition of $\text{RANF}(\hat{Q})$. □

Now we derive a bound on $\left|\left[\!\left[\hat{Q}'\right]\!\right]\right|$ for an arbitrary RANF subquery $\hat{Q}' \sqsubseteq \hat{Q}$, $\hat{Q} \in \{\hat{Q}_{\text{fin}}, \hat{Q}_{\text{inf}}\}$, with respect to Q and $\hat{Q}$.

Lemma D.2. *Let Q be an RC query and let $\text{rw}(Q) = (\hat{Q}_{\text{fin}}, \hat{Q}_{\text{inf}})$. Let $\hat{Q}' \sqsubseteq \hat{Q}$ be a RANF subquery of $\hat{Q} \in \{\hat{Q}_{\text{fin}}, \hat{Q}_{\text{inf}}\}$. Suppose that $\hat{Q}'$ is not a quantified predicate. Then*

$$\left|\left[\!\left[\hat{Q}'\right]\!\right]\right| \cdot |\text{fv}(\hat{Q}')| \leq \sum_{\mathcal{P}^{\exists} \subseteq \mathcal{A}^{\exists}(Q)} \left\| \left[\!\!\left[\bigwedge_{P^{\exists} \in \mathcal{P}^{\exists}} P^{\exists} \right]\!\!\right] \right\| \cdot 2^{|\text{av}(\hat{Q})|^2} \cdot |\text{av}(\hat{Q})|\,.$$

Proof. Applying Lemma D.1 to the RANF subquery $\hat{Q}'$ yields

$$\left|\left[\!\left[\hat{Q}'\right]\!\right]\right| \leq \sum_{\substack{\mathcal{P}^{\exists} \subseteq \mathcal{A}^{\exists}(\hat{Q}') \\ \mathcal{E} \subseteq C(\text{fv}(\mathcal{P}^{\exists}), \text{fv}(\hat{Q}')) \\ \text{fv}(\mathcal{P}^{\exists}) \cup \text{fv}(\mathcal{E}) = \text{fv}(\hat{Q}')}} \left\| \left[\!\!\left[\bigwedge_{P^{\exists} \in \mathcal{P}^{\exists}} P^{\exists} \wedge \bigwedge_{E \in \mathcal{E}} E \right]\!\!\right] \right\|.$$

Next we observe that

$$\left\| \left[\!\!\left[\bigwedge_{P^{\exists} \in \mathcal{P}^{\exists}} P^{\exists} \wedge \bigwedge_{E \in \mathcal{E}} E \right]\!\!\right] \right\| \leq \left\| \left[\!\!\left[\bigwedge_{P^{\exists} \in \mathcal{P}^{\exists}} P^{\exists} \right]\!\!\right] \right\|$$

as equalities $E \in \mathcal{E}$ can only restrict the respective set of tuples and duplicate columns. The number of equalities in $C(\text{fv}(\mathcal{P}^{\exists}), \text{fv}(\hat{Q}'))$, where $\mathcal{P}^{\exists} \in \mathcal{A}^{\exists}(\hat{Q}')$ and $\text{fv}(\mathcal{P}^{\exists}) \subseteq \text{fv}(\hat{Q}')$, is at most

$$|\text{fv}(\mathcal{P}^{\exists})| \cdot |\text{fv}(\hat{Q}')| \leq |\text{fv}(\hat{Q}')|^2 \leq |\text{av}(\hat{Q})|^2\,,$$

where the last inequality holds because the free variables in a subquery $\hat{Q}'$ of a query $\hat{Q}$ are included in the set of all free and bound variables of the query $\hat{Q}$. Hence, the number of subsets $\mathcal{E} \subseteq C(\text{fv}(\mathcal{P}^{\exists}), \text{fv}(\hat{Q}))$ is at most $2^{|\text{av}(\hat{Q})|^2}$.

Finally, for any subquery $\hat{Q}'$ of a query $\hat{Q}$ such that $\hat{Q}'$ is not a quantified predicate, $\mathcal{A}^{\exists}(\hat{Q}') \subseteq \mathcal{A}^{\exists}(\hat{Q})$ holds by well-founded induction over the definition of the function $\mathcal{A}^{\exists}(\hat{Q})$. Combining this with Lemma 4.10 yields $\mathcal{A}^{\exists}(\hat{Q}') \subseteq \mathcal{A}^{\exists}(Q)$. □

We bound the cost of RANF query $\hat{Q} \in \{\hat{Q}_{\text{fin}}, \hat{Q}_{\text{inf}}\}$ as follows:

Lemma D.3. *Let Q be an RC query and let* $\text{rw}(Q) = (\hat{Q}_{\text{fin}}, \hat{Q}_{\text{inf}})$. *Let $\hat{Q} \in \{\hat{Q}_{\text{fin}}, \hat{Q}_{\text{inf}}\}$. Then*

$$\text{cost}(\hat{Q}) \leq |\hat{Q}| \cdot \sum_{\mathcal{P}^\exists \subseteq \mathcal{A}^\exists(Q)} \left|\!\left[\!\left[\bigwedge_{P^\exists \in \mathcal{P}^\exists} P^\exists \right]\!\right]\right| \cdot 2^{|\text{av}(\hat{Q})|^2} \cdot |\text{av}(\hat{Q})| + |\hat{Q}| \cdot \max_{P \in \mathcal{A}(Q)} |\,[\![P]\!]\,| \cdot |\text{av}(\hat{Q})|\,.$$

Proof. Observe that the number of RANF subqueries $\hat{Q}'$ of a RANF query $\hat{Q}$ is bounded by the number of constructors in $\hat{Q}$. We derive the first line of our estimation of $\text{cost}(\hat{Q})$ using Lemma D.2 for all subqueries $\hat{Q}' \sqsubseteq \hat{Q}$ that are not quantified predicates.

If $\hat{Q}' \sqsubseteq \hat{Q}$ is a quantified predicate, then there exists an atomic predicate $P \in \mathcal{A}(Q)$ such that $|\,[\![\hat{Q}']\!]\,| \leq |\,[\![P]\!]\,|$. This justifies the second line of our estimation of $\text{cost}(\hat{Q})$. □

Next we observe that the join of any two quantified predicates $P_1^\exists, P_2^\exists \in \mathcal{A}^\exists(Q)$ of the form $\exists \vec{z}_1.\, P$ and $\exists \vec{z}_2.\, P$, respectively, for some $P \in \mathcal{A}(Q)$, equals one of the two quantified predicates $P_1^\exists$ or $P_2^\exists$. This is because $\vec{z}_1$ is a suffix of $\vec{z}_2$ or vice-versa. We can bound the cardinality of the join $\bigwedge_{P^\exists \in \mathcal{P}^\exists} P^\exists$, where $\mathcal{P}^\exists \subseteq \mathcal{A}^\exists(Q)$, by

$$\left|\!\left[\!\left[\bigwedge_{P^\exists \in \mathcal{P}^\exists} P^\exists \right]\!\right]\right| \leq \prod_{P \in \mathcal{A}(Q)} |\,[\![P]\!]\,|$$

and the cost of a RANF query $\hat{Q} \in \{\hat{Q}_{\text{fin}}, \hat{Q}_{\text{inf}}\}$ is thus at most

$$\text{cost}(\hat{Q}) \leq |\hat{Q}| \cdot 2^{|\mathcal{A}^\exists(Q)|} \cdot \left(\prod_{P \in \mathcal{A}(Q)} |\,[\![P]\!]\,|\right) \cdot 2^{|\text{av}(\hat{Q})|^2} \cdot |\text{av}(\hat{Q})| + |\hat{Q}| \cdot \max_{P \in \mathcal{A}(Q)} |\,[\![P]\!]\,| \cdot |\text{av}(\hat{Q})|\,.$$

Although $|\hat{Q}| \cdot 2^{|\mathcal{A}^\exists(Q)|} \cdot 2^{|\text{av}(\hat{Q})|^2} \cdot |\text{av}(\hat{Q})|$ can be superexponential in $|Q|$, this value only depends on the query Q and thus does not contribute to the data complexity of evaluating $\hat{Q} \in \{\hat{Q}_{\text{fin}}, \hat{Q}_{\text{inf}}\}$.

E IMPLEMENTATION DETAILS

In this section we provide a detailed description of our translation RC2SQL. Overall, the translation is defined as

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

where

$$\begin{aligned} Q'_{\text{fin}} &\coloneqq \text{RANF2SQL}(\text{OPTCNT}(Q_{\text{fin}}))\,,\\ Q'_{\text{inf}} &\coloneqq \text{RANF2SQL}(\text{OPTCNT}(Q_{\text{inf}}))\,,\\ (Q_{\text{fin}}, Q_{\text{inf}}) &\coloneqq \text{rw}(Q). \end{aligned}$$

Function $\text{rw}(\cdot)$ is defined in Section 4.4 as a composition of the $\text{split}(\cdot)$ and $\text{SR2RANF}(\cdot)$ functions, which are defined in Section 4.3 and Appendix C, respectively. Below we first describe how we instantiate the nondeterministic choices in all our algorithms. Then we define functions $\text{OPTCNT}(\cdot)$ and $\text{RANF2SQL}(\cdot)$.

E.1 Instantiating Our Translation

To guide the nondeterministic choices in our algorithms, we suppose that the algorithms have access to a *training database* 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. Nevertheless, our translation satisfies the correctness and worst-case complexity claims (Section 4.3 and 4.4) for any choice of the training database. We describe the training databases used in our empirical evaluation in Section 6. Because of its constant size, the complexity of evaluating any query over the training database is constant and does not impact the data complexity of evaluating an arbitrary query over the actual database using our translation. There are two types of nondeterministic choices in our algorithms:

- Choosing some $X \in \mathcal{X}$ with $P(X)$ in a while-loop with a loop condition $\exists X \in \mathcal{X}.\, P(X)$. As the bodies of such 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 variable $y \in W$ and a set C such that $\text{cov}(y, F, C)$, where F is a query with range-restricted bound variables and $W \subseteq \text{fv}(F)$ is a subset of its free variables. Observe that the measure $\text{sz}(Q)$ on queries, defined in Figure 8, decreases for the queries in the premises of the rules for $\text{gen}(y, F, C)$ and $\text{cov}(y, F, C)$, defined in Figure 1 and 2. Hence, deriving $\text{gen}(y, F, C)$ and $\text{cov}(y, F, C)$ either succeeds or gets stuck after at most $\text{sz}(F)$ steps. In particular, we can enumerate all sets C such that $\text{cov}(y, F, C)$ holds. Because we derive one additional query $F[y \mapsto x]$ for every $x \in \mathcal{E}_y(C)$ and a single query $F \wedge \mathcal{G}^\vee(C)$, we choose $y \in W$ and a set C minimizing $|\mathcal{E}_y(C)|$ as the first objective and $\sum_{P \in \mathcal{G}(C)} \text{cost}(P)$ as the second objective. Our particular choice of C with $\text{cov}(y, F, C)$ is merely a heuristic and does not provide any additional guarantees compared to any other choice of C with $\text{cov}(y, F, C)$.

We remark that one could apply the translations (★) and (★∃) from Lemma 4.4 and Lemma 4.6 also to variables that are range-restricted. This might further improve the performance of evaluating the translated query. But since we do not gain any additional worst-case complexity guarantees, we refrain from applying (★) and (★∃) to range-restricted variables in our translation.

E.2 Optimization using Count Aggregations

In this section, we introduce count aggregations and describe an optimization of RANF queries by evaluating certain subqueries using count aggregations. Consider the RANF query

$$(\exists y.\, S(x, y)) \wedge \neg \exists y.\, ((\exists y.\, S(x, y)) \wedge R(100, y) \wedge \neg S(x, y))\,,$$

computed by our translation for a query from [24, Example 5.4]. This RANF query is an example of the more general pattern

$$\varphi(x) \wedge \neg \exists y.\, (\varphi(x) \wedge \psi(y) \wedge \neg \eta(x, y))\,,$$

which is introduced by applying our translation to the query $\varphi(x) \wedge \forall y.\, (\psi(y) \longrightarrow \eta(x, y))$. The cost of this query is dominated by the cost of the subquery $\varphi(x) \wedge \psi(y)$. Consider the subquery $A \coloneqq \exists y.\, (\varphi(x) \wedge \psi(y) \wedge \neg \eta(x, y))$. A fixed $x = x_0$ satisfies A if $\varphi(x_0)$ holds and the number of values for y with $\psi(y)$ is strictly larger than the number of values for y with $\psi(y)$ that are excluded by $\eta(x_0, y)$, i.e., $|\,[\![\psi(y)]\!]\,| > |\,[\![\psi(y) \wedge \eta(x_0, y)]\!]\,|$. Because $\psi(y) \wedge \eta(x_0, y)$ implies $\psi(y)$, it always holds that $|\,[\![\psi(y)]\!]\,| \geq |\,[\![\psi(y) \wedge \eta(x_0, y)]\!]\,|$ and thus $|\,[\![\psi(y)]\!]\,| > |\,[\![\psi(y) \wedge \eta(x_0, y)]\!]\,|$ if and only if $|\,[\![\psi(y)]\!]\,| \neq |\,[\![\psi(y) \wedge \eta(x_0, y)]\!]\,|$. An alternative evaluation of A filters the set $[\![\varphi(x)]\!]$ by the condition $|\,[\![\psi(y)]\!]\,| \neq |\,[\![\psi(y) \wedge \eta(x_0, y)]\!]\,|$, where $|\,[\![\psi(y)]\!]\,|$ is independent of $x = x_0$ and thus this value can be precomputed. The data complexity of the alternative evaluation is at most the data complexity of the original evaluation and improves if $|\,[\![\varphi(x) \wedge \psi(y) \wedge \eta(x, y)]\!]\,| \ll |\,[\![\varphi(x) \wedge \psi(y)]\!]\,|$. The values $|\,[\![\psi(y)]\!]\,|$ and $|\,[\![\psi(y) \wedge \eta(x_0, y)]\!]\,|$ can be computed using the

count aggregations on the respective queries. Furthermore, we observe that a fixed $x = x_0$ satisfies $\varphi(x) \wedge \neg A$ if $\varphi(x_0)$ holds and $x = x_0$ does not satisfy A, i.e., the number of values for y with $\psi(y)$ is equal to the number of values for y with $\psi(y)$ and $\eta(x_0, y)$, i.e., $|[\![\psi(y)]\!]| = |[\![\psi(y) \wedge \eta(x_0, y)]\!]|$.

Next we introduce the syntax and semantics of count aggregations. We denote a count aggregation query by $[\mathsf{CNT}\,\vec{y}.\,F](s;\vec{g})$, where F is a query, s is a variable, and $\vec{y}, \vec{g}$ are sequences of pairwise distinct variables such that $s \notin \mathrm{fv}(F)$, $\vec{y} \cap \vec{g} = \emptyset$, and $\vec{y} \cup \vec{g} = \mathrm{fv}(F)$ where the sequence $\vec{g}$ are group-by variables, the sequence $\vec{y}$ are aggregated variables, and s is the count for the corresponding group. The semantics of count aggregation queries is defined as follows:

$$(\mathcal{D}, v) \models [\mathsf{CNT}\,\vec{y}.\,F](s;\vec{g}) \text{ iff } (M = \emptyset \longrightarrow \mathrm{fv}(F) \subseteq \vec{y}) \wedge v(s) = |M|,$$

where $M = \{\vec{d} \in \mathbb{D}^{|\vec{y}|} \mid (\mathcal{D}, v[\vec{y} \mapsto \vec{d}]) \models F\}$. We use the condition $M = \emptyset \longrightarrow \mathrm{fv}(F) \subseteq \vec{y}$ instead of $M \neq \emptyset$ to set s to a zero count if there are no group-by variables (like in SQL). The set of free variables in a count aggregation is $\mathrm{fv}([\mathsf{CNT}\,\vec{y}.\,F](s;\vec{g})) = \vec{g} \cup \{s\}$. We also extend the definition of $\mathrm{RANF}(Q)$ with the case of a count aggregation query:

$$\mathrm{RANF}([\mathsf{CNT}\,\vec{y}.\,F](s;\vec{g})) \quad \text{iff} \quad \mathrm{RANF}(F) \wedge s \notin \mathrm{fv}(F) \wedge \vec{y} \cap \vec{g} = \emptyset \wedge \vec{y} \cup \vec{g} = \mathrm{fv}(F)\,.$$

We formulate translations introducing count aggregations in the following two lemmas. Here we define the sequence $\vec{x} := \vec{\mathrm{fv}}(P) \setminus \vec{y}$ which is obtained from the sequence $\vec{\mathrm{fv}}(P)$ of the free variables in P by dropping all elements from the sequence $\vec{y}$.

Lemma E.1. *Let $\exists\vec{y}.\,P \wedge \bigwedge_{i=1}^{n} \neg R_i$, $n \geq 1$, be a RANF query. Let $\vec{x} := \vec{\mathrm{fv}}(P) \setminus \vec{y}$ be the free variables in $\exists\vec{y}.\,P \wedge \bigwedge_{i=1}^{n} \neg R_i$. Let s, s' be fresh variables that do not occur in $\mathrm{fv}(P)$. Then*

$$\begin{aligned}(\exists\vec{y}.\,P \wedge \textstyle\bigwedge_{i=1}^{n} \neg R_i) \equiv ((\exists\vec{y}.\,P) \wedge \textstyle\bigwedge_{i=1}^{n} \neg(\exists\vec{y}.\,P \wedge R_i)) \vee \\ (\exists s\, s'.\, [\mathsf{CNT}\,\vec{y}.\,P](s;\vec{x}) \wedge \\ [\mathsf{CNT}\,\vec{y}.\,\textstyle\bigvee_{i=1}^{n}(P \wedge R_i)](s';\vec{x}) \\ \wedge \neg(s = s'))\,.\end{aligned} \tag{\#}$$

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

Lemma E.2. *Let $S \wedge \neg\exists\vec{y}.\,P \wedge \bigwedge_{i=1}^{n} \neg R_i$, $n \geq 1$, be a RANF query. Let $\vec{x} := \vec{\mathrm{fv}}(P) \setminus \vec{y}$ be the free variables in $\exists\vec{y}.\,P \wedge \bigwedge_{i=1}^{n} \neg R_i$. Let s, s' be fresh variables that do not occur in $\mathrm{fv}(S) \cup \mathrm{fv}(P)$. Then*

$$\begin{aligned}(S \wedge \neg\exists\vec{y}.\,P \wedge \textstyle\bigwedge_{i=1}^{n} \neg R_i) \equiv (S \wedge \neg(\exists\vec{y}.\,P)) \vee (\exists s\, s'.\, S \wedge \\ [\mathsf{CNT}\,\vec{y}.\,P](s;\vec{x}) \wedge \\ [\mathsf{CNT}\,\vec{y}.\,\textstyle\bigvee_{i=1}^{n}(P \wedge R_i)](s';\vec{x}) \\ \wedge (s = s'))\,.\end{aligned} \tag{\#\#}$$

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 $[\mathsf{CNT}\,\vec{y}.\,P](s;\vec{x})$ in which P is evaluated before the count aggregation is computed. For the subquery $A := \exists y.\,(\varphi(x) \wedge \psi(y) \wedge \neg\eta(x, y))$ from before, we would have $P := \varphi(x) \wedge \psi(y)$, i.e., we would not (yet) avoid computing the Cartesian product $\varphi(x) \wedge \psi(y)$. However, we could reduce the scope of the bound variable y by further translating

$$[\mathsf{CNT}\,y.\,\varphi(x) \wedge \psi(y)](s;x) \equiv \varphi(x) \wedge [\mathsf{CNT}\,y.\,\psi(y)](s;[])\,,$$

which corresponds to precomputing $s = |[\![\psi(y)]\!]|$ as discussed before. This technique called *mini-scoping* can be applied to a count aggregation if the aggregated query P is a conjunction that can be split into two conjuncts $P \equiv P_1 \wedge P_2$ such that $\mathrm{fv}(P_1) \cap \vec{y} = \emptyset$, $\mathrm{RANF}(P_1)$, and $\mathrm{RANF}(P_2)$. Then the scope of the count aggregation can be reduced to P_2. Mini-scoping can be analogously applied to queries $\exists\vec{y}.\,P$. Note that mini-scoping can improve the query cost (i.e., data complexity) only if $|[\![P_2]\!]| \ll |[\![P_1 \wedge P_2]\!]|$, e.g., if $\mathrm{fv}(P_1) \cap \mathrm{fv}(P_2) = \emptyset$. We implement the optimizations captured by lemmas E.1 and E.2, as well as the mini-scoping optimization mentioned above in a function called $\mathrm{OPTCNT}(\cdot)$. Given a RANF query Q, $\mathrm{OPTCNT}(Q)$ returns an equivalent optimized RANF query.

Example E.3. We show how we introduce count aggregations into the query

$$Q := \varphi(x) \wedge \neg\exists y.\,(\varphi(x) \wedge \psi(y) \wedge \neg\eta(x, y))\,,$$

abstracting the RANF query computed by our translation for a query from Example 5.4 by Van Gelder and Topor [24]. After applying the translation (##) and mini-scoping to this query, we obtain the following equivalent RANF query:

$$\begin{aligned}\mathrm{OPTCNT}(Q) := (\varphi(x) \wedge \neg(\varphi(x) \wedge \exists y.\,\psi(y))) \vee \\ (\exists s\, s'.\, \varphi(x) \wedge [\mathsf{CNT}\,y.\,\psi(y)](s;[]) \wedge \\ [\mathsf{CNT}\,y.\,\varphi(x) \wedge \psi(y) \wedge \eta(x, y)](s';x) \\ \wedge (s = s'))\,.\end{aligned}$$

E.3 Translating RANF to SQL

Our translation of a RANF query into SQL has two steps: we first translate query to an equivalent RA expression, which we then translate to SQL using a publicly available RA interpreter `radb` [26].

We define the function $\mathrm{RANF2RA}(Q)$ translating RANF queries Q into equivalent RA expressions E_Q. The translation is based on Algorithm 5.4.8 by Abiteboul et al. [2], which we modify as follows.

We first add support for closed RC queries. Chomicki and Toman [9] observed that closed RC queries cannot be handled by SQL, since it does not allow empty projections, nor 0-ary relations. They propose to use a unary auxiliary predicate $t \in R$ whose interpretation $t^{\mathcal{D}} = \{\tau\}$ always contains one tuple (τ). Any closed query $\exists x.\,Q$ is then translated into $\exists x.\,T(t) \wedge Q$ with an auxiliary free variable t. Any other closed query Q is translated into $T(t) \wedge Q$, e.g., $P(42)$ is translated into $T(t) \wedge P(42)$. We also use the predicate t to translate queries of the form $x \approx c$ and $c \approx x$ because its single tuple (τ) can be "projected" to any constant value.

We handle RANF queries of the form $\varphi \wedge x \approx y$, where $x \in \mathrm{fv}(\varphi)$, and $y \notin \mathrm{fv}(\varphi)$ differently. Abiteboul et al. [2] translate them into $\sigma_{x \approx y}(E_\varphi \bowtie \delta_{x \mapsto y} E_\varphi)$ where $\delta_{x \mapsto y}$ denotes renaming the attribute x to y and $\sigma_{x \approx y}$ is the selection operator with the condition $x \approx y$ on the attributes x and y. Alternatively, the query $\varphi \wedge x \approx y$ can be evaluated by *duplicating* the attribute x in E_φ and renaming the duplicated attribute y. We opt for the alternative approach since duplicating a column performs better than performing a natural join and evaluating a selection operator.

Finally, we also handle the count aggregation queries introduced by our optimizations from Appendix E.2.

The `radb` interpreter translates a RA expression into SQL, by simply mapping the RA constructors into their SQL counterparts, captured by the function $\mathrm{RA2SQL}(\cdot)$, which we leave unspecified. The function is primitive recursive on the RA datatype. We modify `radb` to further improve performance of the query evaluation as follows.

A RANF query $\varphi \wedge \neg\psi$, where RANF(φ), RANF(ψ), and $\mathrm{fv}(\psi) \subseteq \mathrm{fv}(\varphi)$ is translated into RA expression E_φ **diff** E_ψ, where **diff** denotes the *generalized* difference operator and E_φ, E_ψ are the equivalent relational algebra expressions for φ, ψ, respectively. `radb` defines the generalized difference operator as E_φ **diff** $E_\psi \coloneqq E_\varphi - (E_\varphi \bowtie E_\psi)$, where $-$ denotes the set difference and $\bowtie$ denotes the natural join. Alternatively, the generalized difference operator can be directly translated into `LEFT JOIN` in SQL. We therefore adjust the implementation of the `diff` operator in `radb` to use `LEFT JOIN` since this translation performs better in our empirical evaluation.

The `radb` interpreter introduces a separate SQL subquery in a `WITH` clause for every subexpression in the input 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 3.3).

Finally, the function RANF2SQL(Q) (Figure 10) is defined as RA2SQL(RANF2RA(Q)), i.e., it composes the above translations.

F DATA GOLF BENCHMARK

We devise the *Data Golf* benchmark for generating synthetic databases. Given a query Q, the generated database guarantees that no $Q' \sqsubseteq Q$ is satisfied by (almost) all possible tuples or (almost) no tuple at all, as this might make Q's evaluation trivial and its benchmarking distorted. We first make the following assumptions on Q:

(1) the bound variable y in every subquery $\exists y.\, F$ of Q satisfies $\mathrm{con}(y, F)$ (see Figure 7), to avoid subqueries like $\exists y.\, \neg P(x, y)$, and every atomic predicate containing y contains a free variable of Q, to avoid subqueries like $\exists y.\, (P(x, y) \vee Q(y))$;
(2) Q contains no closed subqueries because a closed subquery is either satisfied by all possible tuples or no tuple at all; and
(3) Q contains no repeated predicate symbols, to avoid subqueries like $P(x) \wedge \neg P(x)$.

Given a sequence of distinct variables $\vec{m}$ and a tuple $\vec{z}$ of the same length $|\vec{z}| = |\vec{m}|$, we may interpret the tuple $\vec{z}$ as a *tuple over the variables* $\vec{m}$, denoted by $\vec{z}^{\vec{m}}$. Given a sequence $t_1, \ldots, t_l \in \vec{m} \cup C$ of terms, we denote by $\vec{z}^{\vec{m}}[t_1, \ldots, t_l]$ the tuple obtained by evaluating the terms $t_1, \ldots, t_l$ with respect to $\vec{z}^{\vec{m}}$ over $\vec{m}$. Formally, we define $\vec{z}^{\vec{m}}[t_1, \ldots, t_l] \coloneqq (v_i)_{i=1}^{l}$, where v_i is either z_j, if $t_i = \vec{m}_j$ or t_i if $t_i \in C$. We lift this notion to sets of tuples over $\vec{m}$ in the standard way.

Data Golf is loosely inspired by *regex golf* [1], a game whose objective is to write a shortest possible regular expression matching a fixed set of strings and not matching another (disjoint) set of strings. Data Golf is formalized by the function $\mathrm{DG}(Q, \vec{m}, \mathcal{P}^{\vec{m}}, \mathcal{N}^{\vec{m}}, \beta)$ (Figure 14) that computes a structure $\mathcal{D}$ such that $\mathcal{P}^{\vec{m}}[\vec{\mathrm{fv}}(Q)] \subseteq [\![Q]\!]$, $\mathcal{N}^{\vec{m}}[\vec{\mathrm{fv}}(Q)] \cap [\![Q]\!] = \emptyset$, and $|[\![Q']\!]|, |[\![\neg Q']\!]|$ contain at least $\min\{|\mathcal{P}^{\vec{m}}|, |\mathcal{N}^{\vec{m}}|\}$ tuples each, for any subquery $Q' \sqsubseteq Q$, where Q is a query, $\vec{m}$ is a sequence of distinct variables, $\mathrm{fv}(Q) \subseteq \vec{m}$, $\mathcal{P}^{\vec{m}}$ and $\mathcal{N}^{\vec{m}}$ are sets of tuples over $\vec{m}$, and $\beta \in \{0, 1\}$ is a strategy.

The function $\mathrm{DG}(Q, \vec{m}, \mathcal{P}^{\vec{m}}, \mathcal{N}^{\vec{m}}, \beta)$ can fail on an equality between two variables $x \approx y$. In this case, we must reject the query Q. We define the *not-depth* of a subquery $x \approx y$ in Q to be the number of negations on the path between the subquery $x \approx y$ and Q's main connective. To prevent failure, we generate the sets $\mathcal{P}^{\vec{m}}$, $\mathcal{N}^{\vec{m}}$, $Z_1^{\vec{m}}$, and $Z_2^{\vec{m}}$ to only contain tuples with equal values for all variables in equalities with even not-depth and pairwise distinct values for all variables in equalities with odd not-depth. This is not always possible, e.g., for $x \approx y \wedge \neg x \approx y$, in which case we reject the query.

In the case of a conjunction or a disjunction, we add disjoint sets $Z_1^{\vec{m}}$, $Z_2^{\vec{m}}$ of tuples over $\vec{m}$ to $\mathcal{P}^{\vec{m}}$, $\mathcal{N}^{\vec{m}}$ so that the intermediate results for the subqueries are not equal because that would be yet another special case in the evaluation that we seek to avoid. In the strategy $\beta = 1$, we use $Z_1^{\vec{m}} = \mathcal{P}^{\vec{m}}$ and $Z_1^{\vec{m}} = \mathcal{N}^{\vec{m}}$, respectively.

input: A RC query Q with pairwise distinct free and bound variables satisfying (1), (2), and (3) in Appendix F, a sequence of distinct variables $\vec{m}$, $\mathrm{fv}(Q) \subseteq \vec{m}$, sets of tuples $\mathcal{P}^{\vec{m}}$, $\mathcal{N}^{\vec{m}}$ over $\vec{m}$ such that $\mathcal{P}^{\vec{m}}[x] \cap \mathcal{N}^{\vec{m}}[x] = \emptyset$, $|\mathcal{P}^{\vec{m}}[x]| \geq |\mathcal{P}^{\vec{m}}|$, and $|\mathcal{N}^{\vec{m}}[x]| \geq |\mathcal{N}^{\vec{m}}|$, for every $x \in \vec{m}$, $\beta \in \{0, 1\}$.
output: A structure $\mathcal{D}$ such that $\mathcal{P}^{\vec{m}}[\vec{\mathrm{fv}}(Q)] \subseteq [\![Q]\!]$, $\mathcal{N}^{\vec{m}}[\vec{\mathrm{fv}}(Q)] \cap [\![Q]\!] = \emptyset$, and $|[\![Q']\!]|, |[\![\neg Q']\!]|$ contain at least $\min\{|\mathcal{P}^{\vec{m}}|, |\mathcal{N}^{\vec{m}}|\}$ tuples each, for any subquery $Q' \sqsubseteq Q$.

function $\mathrm{DG}(Q, \vec{m}, \mathcal{P}^{\vec{m}}, \mathcal{N}^{\vec{m}}, \beta) =$
  switch Q **do**
    case $r(t_1, \ldots, t_{\iota(r)})$ **do return** $\{r^{\mathcal{D}} \mapsto \mathcal{P}^{\vec{m}}[t_1, \ldots, t_{\iota(r)}]\}$;
    case $x \approx y$ **do**
      if $\exists v_1, v_2.\, (v_1 \neq v_2 \wedge (v_1, v_2) \in \mathcal{P}^{\vec{m}}[x, y]) \vee (v_1 = v_2 \wedge (v_1, v_2) \in \mathcal{N}^{\vec{m}}[x, y])$ **then**
        fail
    case $\neg F$ **do return** $\mathrm{DG}(F, \vec{m}, \mathcal{N}^{\vec{m}}, \mathcal{P}^{\vec{m}}, \beta)$;
    case $F_1 \vee F_2$ *or* $F_1 \wedge F_2$ **do**
      $n \coloneqq \min\{|\mathcal{P}^{\vec{m}}|, |\mathcal{N}^{\vec{m}}|\}$;
      $(Z_1^{\vec{m}}, Z_2^{\vec{m}}) \leftarrow \{(Z_1^{\vec{m}}, Z_2^{\vec{m}}) \mid |Z_1^{\vec{m}}| = |Z_2^{\vec{m}}| = n \wedge \forall x \in \vec{m}.\, Z_1^{\vec{m}}[x] \cap Z_2^{\vec{m}}[x] = \emptyset \wedge (Z_1^{\vec{m}}[x] \cup Z_2^{\vec{m}}[x]) \cap (\mathcal{P}^{\vec{m}}[x] \cup \mathcal{N}^{\vec{m}}[x]) = \emptyset\}$;
      if $\beta = 0$ **then**
        return $\mathrm{DG}(F_1, \vec{m}, \mathcal{P}^{\vec{m}} \cup Z_1^{\vec{m}}, \mathcal{N}^{\vec{m}} \cup Z_2^{\vec{m}}, \beta) \cup \mathrm{DG}(F_2, \vec{m}, \mathcal{P}^{\vec{m}} \cup Z_2^{\vec{m}}, \mathcal{N}^{\vec{m}} \cup Z_1^{\vec{m}}, \beta)$
      else
        switch Q **do**
          case $F_1 \vee F_2$ **do**
            return $\mathrm{DG}(F_1, \vec{m}, \mathcal{P}^{\vec{m}}, \mathcal{N}^{\vec{m}} \cup Z_2^{\vec{m}}, \beta) \cup \mathrm{DG}(F_2, \vec{m}, Z_2^{\vec{m}}, \mathcal{N}^{\vec{m}} \cup \mathcal{P}^{\vec{m}}, \beta)$
          case $F_1 \wedge F_2$ **do**
            return $\mathrm{DG}(F_1, \vec{m}, \mathcal{P}^{\vec{m}} \cup \mathcal{N}^{\vec{m}}, Z_2^{\vec{m}}, \beta) \cup \mathrm{DG}(F_2, \vec{m}, \mathcal{P}^{\vec{m}} \cup Z_2^{\vec{m}}, \mathcal{N}^{\vec{m}}, \beta)$
    case $\exists y.\, F$ **do**
      $\{\vec{z}_1, \ldots, \vec{z}_n\} \coloneqq \mathcal{P}^{\vec{m}}$; $\{\vec{z}'_1, \ldots, \vec{z}'_{n'}\} \coloneqq \mathcal{N}^{\vec{m}}$;
      $\{x_1, \ldots, x_n, x'_1, \ldots, x'_{n'}\} \leftarrow \{X \subseteq \mathbb{D} \mid |X| = n + n'\}$;
      return $\mathrm{DG}(F, \vec{m} \cdot y, \{\vec{z}_1 \cdot x_1, \ldots, \vec{z}_n \cdot x_n\}, \{\vec{z}'_1 \cdot x'_1, \ldots, \vec{z}'_{n'} \cdot x'_{n'}\}, \beta)$

Figure 14: Computing the Data Golf structure.