Practical Relational Calculus Query Evaluation

Martin Raszyk
Department of Computer Science, ETH Zürich, Switzerland

David Basin
Department of Computer Science, ETH Zürich, Switzerland

Srđan Krstić
Department of Computer Science, ETH Zürich, Switzerland

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

Abstract

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

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

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

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

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

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

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

This query is domain-independent and follows closely the informal description. It is not clear how to evaluate it though because its second conjunct is domain-dependent as it is satisfied

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:30
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

for every brand that does not occur in P. Finding suspicious brands using RA or SQL is a challenge, which only the best students from an undergraduate database course will accomplish. We give away an RA answer next (where $-$ is the set difference operator and $\triangleright$ is the anti-join):

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

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

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

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

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

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

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

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

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

After the more standard transformations to relational algebra normal form and from there to RA expressions, we translate the resulting RA expressions into SQL using the radb tool [29]. Along the way to SQL, we leverage different ideas from the literature to optimize the overall result (Section 5). For example, we generalize Claußen et al. [9]'s approach to avoid evaluating Cartesian products like $\pi_{user,score}(\mathsf{S}) \times \mathsf{P}$ in the above translation by using count aggregations.

The overall translation allows us to use standard database management systems to evaluate RC queries. We implement our translation and use PostgreSQL to evaluate the resulting queries. With this setup, we empirically show that our approach outperforms Van Gelder and Topor's translation (which also uses PostgreSQL for evaluation) and other approaches (Section 5). In our evaluation, we use the realistic Amazon review dataset [23] and design a synthetic benchmark that generates hard database instances for random RC queries.

In summary, the following are our three main contributions:

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

2 Related Work

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

Data complexity [28] is the complexity of recognizing if a tuple satisfies a fixed query over a database, as a function of the database size. Our capturability algorithm provides an upper bound on the data complexity for RC queries over an infinite domain that have a finite evaluation result (but it cannot decide if a tuple belongs to a query's result if the result is infinite).

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

Structure reduction. The classical approach to handling arbitrary RC queries is to evaluate them under a finite structure [18]. The core question here is whether the evaluation produces the same result as defined by the natural semantics, which typically considers infinite domains. Codd's theorem [10] affirmatively answers this question for domain-independent queries, restricting the structure to the *active domain*. Ailamazyan et al. [2] show that by extending the active domain with a few additional elements, whose number depends only on the query, one can decide relative safety for RC. Further extensions are *natural–active collapse* results that combine the structure reduction with a translation-based approach [5,15]. Our work is inspired by these theoretical landmarks (Section 4.1). Yet we avoid using (extended) active domains, which make query evaluation impractical due to a prohibitively high time complexity.

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

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

3 Preliminaries

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

3.1 Relational Calculus

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

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

Here, $r \in \mathcal{R}$ is a predicate symbol, $t, t_1, \ldots, t_{\iota(r)} \in \mathcal{V} \cup \mathcal{C}$ are terms, and $x \in \mathcal{V}$ is a variable. We write $\exists \vec{v}.\, Q$ 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 point out that defining $Q_1 \vee Q_2$ as a shorthand for $\neg(\neg Q_1 \wedge \neg Q_2)$ would complicate the definitions of the relevant classes of RC queries, e.g., the safe-range queries.

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

A structure $\mathcal{S}$ over a signature $(\mathcal{C}, \mathcal{R}, \iota)$ consists of a non-empty domain $\mathcal{D}$ and interpretations $\mathsf{c}^{\mathcal{S}} \in \mathcal{D}$ and $r^{\mathcal{S}} \subseteq \mathcal{D}^{\iota(r)}$, for each $\mathsf{c} \in \mathcal{C}$ and $r \in \mathcal{R}$. We assume that all the relations $r^{\mathcal{S}}$ are *finite*. Note that this assumption does *not* yield a finite structure (as defined in finite model theory [18]) since the domain $\mathcal{D}$ can still be infinite. A (*variable*) *assignment* is a mapping $\alpha : \mathcal{V} \to \mathcal{D}$. We additionally define α on constant symbols $\mathsf{c} \in \mathcal{C}$ as $\alpha(\mathsf{c}) = \mathsf{c}^{\mathcal{S}}$. We write $\alpha[x \mapsto d]$ for the assignment that maps x to $d \in \mathcal{D}$ and is otherwise identical to α. We lift this notation to sequences $\vec{x}$ and $\vec{d}$ of pairwise distinct variables and arbitrary domain elements of

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

Figure 1 The semantics of RC.

the same length. The semantics of RC queries for a structure $\mathcal{S}$ and an assignment α is defined in Figure 1. We write $\alpha \models Q$ for $(\mathcal{S},\alpha) \models Q$ if the structure $\mathcal{S}$ is fixed in the given context. For a fixed $\mathcal{S}$, only the assignments to Q's free variables influence $\alpha \models Q$, i.e., $\alpha \models Q$ is equivalent to $\alpha' \models Q$, for every variable assignment α' that agrees with α on $\mathsf{fv}(Q)$. For closed queries Q, we write $\models Q$ and say that Q holds, since closed queries either hold for all variable assignments or for none of them. We call a finite sequence $\vec{d}$ of domain elements $d_1,\dots d_k \in \mathcal{D}$ a *tuple*. Given a query Q and a structure $\mathcal{S}$, we denote the set of satisfying tuples for Q by

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

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

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

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

A query Q is *domain-independent* if $[\![Q]\!]^{\mathcal{S}_1} = [\![Q]\!]^{\mathcal{S}_2}$ holds for every two structures $\mathcal{S}_1$ and $\mathcal{S}_2$ that agree on the interpretations of constants ($\mathsf{c}^{\mathcal{S}_1} = \mathsf{c}^{\mathcal{S}_2}$) and predicates ($r^{\mathcal{S}_1} = r^{\mathcal{S}_2}$), while their domains $\mathcal{D}_1$ and $\mathcal{D}_2$ may differ. Agreement on the interpretations implies $\mathsf{adom}^{\mathcal{S}_1}(Q) = \mathsf{adom}^{\mathcal{S}_2}(Q) \subseteq \mathcal{D}_1 \cap \mathcal{D}_2$. It is undecidable whether an RC query is domain-independent [24,27].

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 $(x \approx x) \equiv \top$, $Q \wedge \bot \equiv \bot$, $Q \vee \bot \equiv Q$, etc. We lift this notation to sets of queries in the standard way. Finally, we denote by $Q[x/\bot]$ the query obtained from Q after replacing every atomic predicate or equality containing a free variable x by $\bot$ (except for $x \approx x$) and performing constant propagation.

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

3.2 Safe-Range Queries

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

We formalize range-restricted variables using the *generated* relation $\mathsf{gen}(x, Q, \mathcal{G})$, defined in Figure 2. Specifically, $\mathsf{gen}(x, Q, \mathcal{G})$ holds if x is a range-restricted variable in Q and every

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

Figure 2 The *generated* relation.

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

Figure 3 The *covered* relation.

satisfying assignment for Q satisfies some quantified predicate, referred to as *generator*, from $\mathcal{G}$. Note that, unlike in a similar definition by Van Gelder and Topor [14, Fig. 5], the rule $\mathsf{gen}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G})$ existentially quantifies the bound variable y from all queries in $\mathcal{G}$ where y occurs. Hence, $\mathsf{gen}(x, Q, \mathcal{G})$ implies $\mathsf{fv}(\mathcal{G}) \subseteq \mathsf{fv}(Q)$. Now we formalize these relationships.

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

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

Relational algebra normal form (RANF) is a class of safe-range queries that can be easily mapped to RA [1]. In Appendix C, we define the predicate $\mathsf{ranf}(\cdot)$ characterizing RANF queries and the translation $\mathsf{sr2ranf}(\cdot)$ of a safe-range query into an equivalent RANF query.

3.3 Query Cost

To assess the time complexity of evaluating a RANF query Q, we define the *cost* of Q over a structure $\mathcal{S}$, denoted $\mathsf{cost}^{\mathcal{S}}(Q)$, to be the sum of intermediate result sizes over all RANF subqueries of Q. Formally, $\mathsf{cost}^{\mathcal{S}}(Q) \coloneqq \sum_{Q' \sqsubseteq Q,\ \mathsf{ranf}(Q')} \left|[\![Q']\!]^{\mathcal{S}}\right| \cdot |\mathsf{fv}(Q')|$. This corresponds to evaluating Q following its RANF structure (Appendix C, Figure 11) using RA operations projection, column duplication, selection, set union, binary join, and anti-join. The complexity of these operations is linear in the combined input and output size (ignoring logarithmic factors due to set operations). The output size (the number of tuples times the number of variables) is counted in $\left|[\![Q']\!]^{\mathcal{S}}\right| \cdot |\mathsf{fv}(Q')|$ and the input size is counted as the output size for the input subqueries. Repeated subqueries are only considered once, which does not affect the asymptotics of query cost. In practice, the evaluation results for common subqueries can be reused.

4 Query Translation

Our approach to evaluating an arbitrary RC query Q over a fixed structure $\mathcal{S}$ with an infinite domain $\mathcal{D}$ proceeds by translating Q into a pair of safe-range queries (Q_{fin}, Q_{inf}) such that

(FV) $\mathsf{fv}(Q_{fin}) = \mathsf{fv}(Q)$ unless Q_{fin} is syntactically equal to $\bot$; $\mathsf{fv}(Q_{inf}) = \emptyset$;

(EVAL) $[\![Q]\!] = [\![Q_{fin}]\!]$ is a finite set if Q_{inf} does not hold; $[\![Q]\!]$ is an infinite set if Q_{inf} holds.

Since the queries Q_{fin} and Q_{inf} are safe-range, they are domain-independent and thus $[\![Q_{fin}]\!]$ is a finite set of tuples. In particular, $[\![Q]\!]$ is a finite set of tuples if Q_{inf} does not hold. In our translation, we combine Hull and Su's case distinction [15] to restrict variables and Van Gelder and Topor's idea [14] to replace the active domain by a smaller set specific to each variable.

4.1 Restricting One Variable

Let $\tilde{Q}$ be a free variable in a query $\tilde{Q}$ with range-restricted bound variables. This assumption on $\tilde{Q}$ will be established by translating an arbitrary query Q bottom-up (Section 4.2). In this section, we develop a translation of $\tilde{Q}$ into an equivalent query $\tilde{Q}'$ so that x is range-restricted except in a single subquery of $\tilde{Q}'$ that is the negation of a disjunction of quantified predicates with a free occurrence of x and equalities of the form $x \approx y$ for some y. From the case distinction "for the corresponding variable: in or out of *adom*, and equality or inequality to other 'previous' variables if out of *adom*" [15], we translate $\tilde{Q}$ into the following equivalent query:

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

Here, $x \in \mathsf{adom}(\tilde{Q})$ stands for an RC query expressing that the single free variable x is in $\mathsf{adom}^{\mathcal{S}}(\tilde{Q})$. The translation distinguishes the following three cases for a fixed valuation α:

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

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

$$\neg \exists \vec{\mathsf{fv}}(\mathcal{G}) \setminus \{x\}.\, \mathcal{G} \Longrightarrow Q \equiv Q[x/\bot] \quad (\text{VGT}_1) \quad \text{and} \quad Q[x/\bot] \Longrightarrow \forall x.\, Q \quad (\text{VGT}_2).$$

Note that $Q[x/\bot] \Longrightarrow \forall x.\, Q$ does not hold for the query $Q \coloneqq \neg \mathsf{B}(x)$ and thus a generator set $\mathcal{G}$ of atomic predicates satisfying (VGT_2) only exists for a proper subset of all RC queries. In contrast, we only require that $\mathcal{G}$ satisfies (VGT_1) in our translation. To this end, we define a *covered* relation $\mathsf{cov}(x, Q, \mathcal{G})$ (in contrast to Van Gelder and Topor's *constrained* relation $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ defined in Appendix A, Figure 8) such that, for every variable x and query $\tilde{Q}$ with range-restricted bound variables, there exists at least one set $\mathcal{G}$ such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ and (VGT_1) holds. Figure 3 shows the definition of this relation. Unlike the generator set $\mathcal{G}$ in $\mathsf{gen}(x, Q, \mathcal{G})$, the *cover* set $\mathcal{G}$ in $\mathsf{cov}(x, Q, \mathcal{G})$ may also contain equalities between two variables. Hence, we define a function $\mathsf{qps}(\mathcal{G})$ that collects all *generators*, i.e., quantified predicates and a function $\mathsf{eqs}(x,\mathcal{G})$ that collects all *variables* y distinct from x occurring in equalities of the form $x \approx y$. We use $\mathsf{qps}^{\vee}(\mathcal{G})$ to denote the query $\bigvee_{Q_{qp} \in \mathsf{qps}(\mathcal{G})} Q_{qp}$. We state the soundness and completeness of the relation $\mathsf{cov}(x,Q,\mathcal{G})$ in the following lemma.

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

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

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

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

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

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

4.2 Restricting Bound Variables

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

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

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

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

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

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

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

Our approach for restricting all bound variables recursively applies Lemma 5. Because the set $\mathcal{G}$ such that $\mathsf{cov}(x,Q,\mathcal{G})$ holds is not necessarily unique, we introduce the following (general) notation. We denote the non-deterministic choice of an object X from a non-empty set $\mathcal{X}$ as $X \leftarrow \mathcal{X}$. We define the recursive function $\mathsf{rb}(Q)$ in Figure 4, 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

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

1 **function** $\mathsf{fixbound}(\mathcal{Q}, x) =$
  $\{Q_{\mathit{fix}} \in \mathcal{Q} \mid x \in \mathsf{nongens}(Q_{\mathit{fix}})\}$;
2 **function** $\mathsf{rb}(Q) =$
3 **switch** Q **do**
4   **case** $\neg Q'$ **do return** $\neg\mathsf{rb}(Q')$;
5   **case** $Q'_1 \vee Q'_2$ **do return** $\mathsf{rb}(Q'_1) \vee \mathsf{rb}(Q'_2)$;
6   **case** $Q'_1 \wedge Q'_2$ **do return** $\mathsf{rb}(Q'_1) \wedge \mathsf{rb}(Q'_2)$;
7   **case** $\exists x.\, Q_x$ **do**
8    $\mathcal{Q} \coloneqq \mathsf{flat}^{\vee}(\mathsf{rb}(Q_x))$;
9   **while** $\mathsf{fixbound}(\mathcal{Q}, x) \neq \emptyset$ **do**
10     $Q_{\mathit{fix}} \leftarrow \mathsf{fixbound}(\mathcal{Q}, x)$;
11     $\mathcal{G} \leftarrow \{\mathcal{G} \mid \mathsf{cov}(x, Q_{\mathit{fix}}, \mathcal{G})\}$;
12     $\mathcal{Q} \coloneqq (\mathcal{Q} \setminus \{Q_{\mathit{fix}}\}) \cup \{Q_{\mathit{fix}} \wedge \mathsf{qps}^{\vee}(\mathcal{G})\} \cup \bigcup_{y \in \mathsf{eqs}(x,\mathcal{G})} \{Q_{\mathit{fix}}[x \mapsto y]\} \cup \{Q_{\mathit{fix}}[x/\bot]\}$;
13   **return** $\bigvee_{\tilde{Q} \in \mathcal{Q}} \tilde{\exists} x.\, \tilde{Q}$;
14   **otherwise do return** Q;

Figure 4 Restricting bound variables.

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

1 **function** $\mathsf{fixfree}(\mathcal{Q}_{\mathit{fin}}) =$
  $\{(Q_{\mathit{fix}}, Q^{=}) \in \mathcal{Q}_{\mathit{fin}} \mid \mathsf{nongens}(Q_{\mathit{fix}}) \neq \emptyset\}$;
2 **function** $\mathsf{inf}(\mathcal{Q}_{\mathit{fin}}, Q) = \{(Q_{\not\infty}, Q^{=}) \in \mathcal{Q}_{\mathit{fin}} \mid \mathsf{disjointvars}(Q_{\not\infty}, Q^{=}) \neq \emptyset \vee \mathsf{fv}(Q_{\not\infty} \wedge Q^{=}) \neq \mathsf{fv}(Q)\}$;
3 **function** $\mathsf{split}(Q) =$
4 $\mathcal{Q}_{\mathit{fin}} \coloneqq \{(\mathsf{rb}(Q), \top)\}$, $\mathcal{Q}_{\mathit{inf}} \coloneqq \emptyset$;
5 **while** $\mathsf{fixfree}(\mathcal{Q}_{\mathit{fin}}) \neq \emptyset$ **do**
6   $(Q_{\mathit{fix}}, Q^{=}) \leftarrow \mathsf{fixfree}(\mathcal{Q}_{\mathit{fin}})$;
7   $x \leftarrow \mathsf{nongens}(Q_{\mathit{fix}})$;
8   $\mathcal{G} \leftarrow \{\mathcal{G} \mid \mathsf{cov}(x, Q_{\mathit{fix}}, \mathcal{G})\}$;
9   $\mathcal{Q}_{\mathit{fin}} \coloneqq (\mathcal{Q}_{\mathit{fin}} \setminus \{(Q_{\mathit{fix}}, Q^{=})\}) \cup \{(Q_{\mathit{fix}} \wedge \mathsf{qps}^{\vee}(\mathcal{G}), Q^{=})\} \cup \bigcup_{y \in \mathsf{eqs}(x,\mathcal{G})} \{(Q_{\mathit{fix}}[x \mapsto y], Q^{=} \wedge x \approx y)\}$;
10   $\mathcal{Q}_{\mathit{inf}} \coloneqq \mathcal{Q}_{\mathit{inf}} \cup \{Q_{\mathit{fix}}[x/\bot]\}$;
11 **while** $\mathsf{inf}(\mathcal{Q}_{\mathit{fin}}, Q) \neq \emptyset$ **do**
12   $(Q_{\not\infty}, Q^{=}) \leftarrow \mathsf{inf}(\mathcal{Q}_{\mathit{fin}}, Q)$;
13   $\mathcal{Q}_{\mathit{fin}} \coloneqq \mathcal{Q}_{\mathit{fin}} \setminus \{(Q_{\not\infty}, Q^{=})\}$;
14   $\mathcal{Q}_{\mathit{inf}} \coloneqq \mathcal{Q}_{\mathit{inf}} \cup \{Q_{\not\infty} \wedge Q^{=}\}$;
15 **return** $(\bigvee_{(Q_{\not\infty}, Q^{=}) \in \mathcal{Q}_{\mathit{fin}}} (Q_{\not\infty} \wedge Q^{=}),\ \mathsf{rb}(\bigvee_{Q_\infty \in \mathcal{Q}_{\mathit{inf}}} \exists \vec{\mathsf{fv}}(Q_\infty).\, Q_\infty))$;

Figure 5 Restricting free variables.

case $\exists x.\, Q_x$. First, $\mathsf{rb}(Q_x)$ is recursively applied on Line 8 to establish the precondition of Lemma 5 that the translated query has range-restricted bound variables. Because existential quantification distributes over disjunction, we flatten disjunction in $\mathsf{rb}(Q_x)$ and process the individual disjuncts independently. We apply (★∃) to every disjunct Q_{fix} in which the variable x is not already range-restricted. For every Q'_{fix} added to $\mathcal{Q}$ after applying (★∃) to Q_{fix} the variable x is either range-restricted or does not occur in Q'_{fix}, i.e., $x \notin \mathsf{nongens}(Q'_{\mathit{fix}})$. This fact entails the termination of the loop on Lines 9–12.

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

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

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

4.3 Restricting Free Variables

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

▶ **Lemma 7.** *Let a structure $\mathcal{S}$ with an infinite domain $\mathcal{D}$ be fixed. Let x be a free variable in a query $\tilde{Q}$ with range-restricted bound variables and let $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ for a set of quantified predicates and equalities $\mathcal{G}$. If $\tilde{Q}[x/\bot]$ is never satisfied, then*

$$\tilde{Q} \equiv (\tilde{Q} \land \mathsf{qps}^{\lor}(\mathcal{G})) \lor \bigvee_{y \in \mathsf{eqs}(x,\mathcal{G})} (\tilde{Q}[x \mapsto y] \land x \approx y), \qquad (☆)$$

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

Proof. If $\tilde{Q}[x/\bot]$ is never satisfied, then (☆) follows from (★). If $\tilde{Q}[x/\bot]$ is satisfied by some tuple, then the last disjunct in (★) applied to $\tilde{Q}$ is satisfied by infinitely many tuples obtained by assigning x some value from the infinite domain $\mathcal{D}$ such that $x \neq y$ holds for all finitely many $y \in \mathsf{eqs}(x, \mathcal{G})$ and x does not appear among the finitely many values interpreting the quantified predicates from $\mathsf{qps}(\mathcal{G})$. ◀

Our approach is implemented by the function $\mathsf{split}(Q)$ defined in Figure 5. In the following, we describe this function and informally justify its correctness formalized by the input/output specification. In $\mathsf{split}(Q)$, we represent the queries Q_{fin} and Q_{inf} using a set $\mathcal{Q}_{\mathit{fin}}$ of query pairs and a set $\mathcal{Q}_{\mathit{inf}}$ of queries such that

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

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

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

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

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

▶ **Lemma 9.** *Let a structure $\mathcal{S}$ with an infinite domain $\mathcal{D}$ be fixed. Let Q be an RC query and $\mathsf{split}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$. If $\not\models Q_{\mathit{inf}}$, then $[\![Q]\!] = [\![Q_{\mathit{fin}}]\!]$ is a finite set. If $\models Q_{\mathit{fin}}$, then $[\![Q]\!]$ is an infinite set.*

▶ **Example 10.** Consider the query $Q^{susp}_{user} \coloneqq \mathsf{B}(b) \wedge \exists s.\, \forall p.\, \mathsf{P}(b,p) \longrightarrow \mathsf{S}(p,u,s)$ from Section 1. Restricting its bound variables yields the query $\mathsf{rb}(Q^{susp}_{user}) = \mathsf{B}(b) \wedge ((\exists s.\, (\neg \exists p.\, \mathsf{P}(b,p) \wedge \neg \mathsf{S}(p,u,s)) \wedge (\exists p.\, \mathsf{S}(p,u,s))) \vee (\neg \exists p.\, \mathsf{P}(b,p)))$ derived in Example 6. Splitting Q^{susp}_{user} yields

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

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

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

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

▶ **Definition 11.** *Let Q be an RC query and $\mathsf{split}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$. Let $\hat{Q}_{\mathit{fin}} \coloneqq \mathsf{sr2ranf}(Q_{\mathit{fin}})$, $\hat{Q}_{\mathit{inf}} \coloneqq \mathsf{sr2ranf}(Q_{\mathit{inf}})$ be the equivalent RANF queries. We define $\mathsf{rw}(Q) \coloneqq (\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}})$.*

4.4 Complexity Analysis

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

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

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

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

When restricting a variable by $\mathsf{qps}^{\vee}(\mathcal{G})$, we only introduce quantified predicates $Q_{qp} \in \overline{\mathsf{qps}}(Q)$.

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

▶ **Theorem 12.** *Let Q be a fixed RC query with pairwise distinct (free and bound) variables. The time complexity of capturing Q, i.e., checking if $[\![Q]\!]$ is finite and enumerating $[\![Q]\!]$ if it is finite, is in* $\tilde{\mathcal{O}}\left(\sum_{\mathcal{Q}_{qps} \subseteq \overline{\mathsf{qps}}(Q), \mathsf{minimal}(\mathcal{Q}_{qps})} \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]|\right)$.

We prove Theorem 12 in Appendix D. Examples 13 and 14 show that the time complexity from Theorem 12 can be achieved neither for the translation of Van Gelder and Topor [14] nor for finite domains. Example 15 shows how equalities affect the bound in Theorem 12.

▶ **Example 13.** Consider the query $Q \coloneqq \mathsf{B}(b) \wedge \exists u, s.\, \neg\exists p.\, \mathsf{P}(b, p) \wedge \neg\mathsf{S}(p, u, s)$, equivalent to Q^{susp} from Section 1. Then $\mathsf{aps}(Q) = \{\mathsf{B}(b), \mathsf{P}(b, p), \mathsf{S}(p, u, s)\}$ and $\overline{\mathsf{qps}}(Q) = \{\mathsf{B}(b), \mathsf{P}(b, p), \exists p.\, \mathsf{P}(b, p), \mathsf{S}(p, u, s), \exists p.\, \mathsf{S}(p, u, s), \exists s, p.\, \mathsf{S}(p, u, s), \exists u, s, p.\, \mathsf{S}(p, u, s)\}$. The translated query Q_{vgt} by Van Gelder and Topor [14] restricts the variables r and s by $\exists s, p.\, \mathsf{S}(p, u, s)$ and $\exists u, p.\, \mathsf{S}(p, u, s)$, respectively. Computing the join of $\exists s, p.\, \mathsf{S}(p, u, s)$ and $\exists u, p.\, \mathsf{S}(p, u, s)$, which is a Cartesian product, results in a time complexity $\Omega(|[\![\mathsf{S}(p, u, s)]\!]|^2)$ for Q_{vgt}. Finally, we observe that $|[\![\mathsf{S}(p, u, s)]\!]|^2$ can be arbitrarily larger than the bound in Theorem 12:

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

▶ **Example 14.** The query $\neg\mathsf{S}(x, y, z)$ is satisfied by a finite set of tuples over a finite domain $\mathcal{D}$ (as is every other query over a finite domain). The number of satisfying tuples is

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

although the bound in Theorem 12 is $\tilde{\mathcal{O}}(|[\![\mathsf{S}(x, y, z)]\!]|)$.

▶ **Example 15.** Consider the following query over the domain $\mathcal{D} = \mathbb{N}$ of natural numbers:

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

Note that this query is equivalent to $Q \equiv \mathsf{B}(x) \wedge \mathsf{B}(y) \wedge \mathsf{B}(z)$ and thus it is satisfied by a finite set of tuples of size $|[\![\mathsf{B}(x)]\!]| \cdot |[\![\mathsf{B}(y)]\!]| \cdot |[\![\mathsf{B}(z)]\!]| = |[\![\mathsf{B}(x)]\!]|^3$. The set of atomic predicates of Q is $\mathsf{aps}(Q) = \{\mathsf{B}(v)\}$ and it must be closed under the equalities occurring in Q to yield a valid bound in Theorem 12. In this case, $\overline{\mathsf{qps}}(Q) = \{\mathsf{B}(v), \exists v.\, \mathsf{B}(v), \mathsf{B}(x), \mathsf{B}(y), \mathsf{B}(z)\}$ and the bound in Theorem 12 is $|[\![\mathsf{B}(v)]\!]| \cdot |[\![\mathsf{B}(x)]\!]| \cdot |[\![\mathsf{B}(y)]\!]| \cdot |[\![\mathsf{B}(z)]\!]| = |[\![\mathsf{B}(x)]\!]|^4$. In particular, the bound is not tight, but it still reflects the complexity of evaluating the RANF queries produced by our translation as it does not derive the equivalence $Q \equiv \mathsf{B}(x) \wedge \mathsf{B}(y) \wedge \mathsf{B}(z)$.

5 Implementation and Empirical Evaluation

We have implemented our translation RC2SQL consisting of roughly 1000 lines of OCaml code [25]. Although our translation satisfies the worst-case complexity bound (Theorem 12), we further improve its average-case complexity by implementing the following optimizations.

- We use a sample structure of constant size, called a *training database*, to estimate the query cost when resolving the nondeterministic choices in our algorithms (Appendix E.1). A good training database should preserve the relative ordering of queries by their cost over the actual database as much as possible. Nevertheless, our translation satisfies the correctness and worst-case complexity claims (Section 4.3 and 4.4) for every choice of the training database. Later we describe the training databases used in our empirical evaluation.
- We use the function optcnt optimizing RANF subqueries of the form $\exists \vec{y}.\, Q^+ \wedge \bigwedge_{i=1}^{k} \neg Q_i^-$ using the count aggregation operator (Appendix E.2). Inspired by Claußen et al. [9], we compare the number of valuations of $\vec{y}$ that satisfy Q^+ and $\bigvee_{i=1}^{k}(Q^+ \wedge Q_i^-)$, respectively.
- To compute an SQL query from a RANF query, we define the function $\mathsf{ranf2sql}(\cdot)$ (Appendix E.3). We first obtain an equivalent RA expression using the standard approach [1] but adjusting the case of closed queries [8]. To translate RA expressions into SQL, we reuse a publicly available RA interpreter radb [29]. We modify its implementation to improve the performance of the resulting SQL query. We map the anti-join operator $\hat{Q}_1 \triangleright \hat{Q}_2$ to a more efficient LEFT JOIN, if $\mathsf{fv}(\hat{Q}_2) \subsetneq \mathsf{fv}(\hat{Q}_1)$, and we perform common subquery elimination.

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

To validate our translation's improved asymptotic time complexity, we compare it with the translation by Van Gelder and Topor [14] (VGT), an implementation of the algorithm by Ailamazyan et al. [2] that uses an extended active domain as the generators, and the DDD [20,21], LDD [7], and MonPoly$^{\mathsf{REG}}$ [4] tools that support direct RC query evaluation using binary decision diagrams. We could not find a publicly available implementation of Van Gelder and Topor's translation. Therefore, the tool VGT for evaluable RC queries is derived from our implementation by modifying the function $\mathsf{rb}(\cdot)$ in Figure 4 to use the relation $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ (Appendix A, Figure 8) instead of $\mathsf{cov}(x, Q, \mathcal{G})$ (Figure 3) and to use the generator $\exists \vec{\mathsf{fv}}(Q) \setminus \{x\}.\, \mathsf{qps}^{\vee}(\mathcal{G})$ instead of $\mathsf{qps}^{\vee}(\mathcal{G})$. Evaluable queries Q are always translated into $(Q_{\mathit{fin}}, \bot)$ by $\mathsf{rw}(\cdot)$ because all of Q's free variables are range-restricted. We also consider translation variants that omit the count aggregation optimization $\mathsf{optcnt}(\cdot)$, marked with a minus ($^-$).

SQL queries computed by the translations are evaluated using the PostgreSQL database engine. We have also used the MySQL database engine but 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 [25].

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 [11] (hence the name) and aims to find a structure on which the result of a given query contains a given *positive* set of tuples and does not contain any tuples from another given *negative* set. The second objective is to ensure that all the query's subqueries evaluate to a non-trivial result. Formally, given a query Q and two sets of tuples $\mathcal{T}^+$ and $\mathcal{T}^-$, representing valuations of $\mathsf{fv}(Q)$, Data Golf produces a pseudorandom structure $\mathcal{S}$, such that $\mathcal{T}^+ \subseteq [\![Q]\!]$, $\mathcal{T}^- \cap [\![Q]\!] = \emptyset$, and $|[\![Q']\!]|$, $|[\![\neg Q']\!]|$ contain at least $\min\{|\mathcal{T}^+|, |\mathcal{T}^-|\}$ tuples each, for every $Q' \sqsubseteq Q$. Data Golf assumes that Q has only constrained [14] bound variables whose generators also contain other variables, i.e., the bound variable y in every subquery $\exists y.\, Q_y$

Experiment SMALL, Evaluable pseudorandom queries Q, $\|\mathsf{sub}(Q)\| = 14$, $n = 500$:										
RC2SQL	**0.3**	**0.2**	**0.3**	**0.3**	**0.2**	**0.3**	**0.3**	0.3	**0.2**	**0.3**
RC2SQL$^-$	**0.3**	**0.2**	148.4	**0.3**	0.3	**0.3**	**0.3**	**0.2**	5.9	**0.3**
VGT	31.2	6.7	4.2	2.5	37.5	9.3	2.4	2.3	11.3	2.7
VGT$^-$	35.1	4.8	117.5	6.3	11.2	21.9	31.6	11.4	12.2	21.9
DDD	10.1	2.5	RE	6.4	5.8	RE	5.8	RE	2.5	5.8
LDD	62.4	23.6	166.8	39.6	53.8	38.8	63.7	TO	17.0	59.1
MonPoly$^{\mathsf{REG}}$	63.9	31.1	142.9	57.6	67.5	54.5	72.2	172.2	33.5	71.4
Experiment MEDIUM, Evaluable pseudorandom queries Q, $\|\mathsf{sub}(Q)\| = 14$, $n = 20000$:										
RC2SQL	2.6	1.4	**3.9**	2.1	**1.5**	2.8	3.3	**1.6**	**1.2**	2.6
RC2SQL$^-$	**2.0**	**1.0**	TO	**2.0**	1.7	**2.2**	**2.3**	1.7	TO	**1.8**
VGT	TO	TO	7.8	3.9	TO	TO	4.8	4.7	TO	4.9
VGT$^-$	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Experiment LARGE, Evaluable pseudorandom queries Q, $\|\mathsf{sub}(Q)\| = 14$, tool = RC2SQL:										
$n = 40000$	3.5	2.7	8.0	4.1	3.0	5.4	6.5	3.6	1.9	5.5
$n = 80000$	7.5	5.4	16.1	8.0	6.0	11.2	13.8	8.2	4.1	11.5
$n = 120000$	10.6	8.2	25.3	11.5	8.9	16.5	20.6	10.7	6.2	16.2
Experiment INFINITE, Non-evaluable pseudorandom queries Q, $\|\mathsf{sub}(Q)\| = 7$, $n = 4000$:										
	Infinite results ($\gamma = 0$)					Finite results ($\gamma = 1$)				
RC2SQL	**0.1**	**0.1**	0.8	0.8	**0.2**	1.0	1.1	0.9	**2.3**	**0.9**
RC2SQL$^-$	**0.1**	**0.1**	**0.5**	**0.5**	**0.2**	**0.7**	**0.7**	**0.6**	TO	1.6
DDD	37.0	37.2	45.4	117.5	78.6	85.9	44.4	50.3	95.5	83.6
LDD	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
MonPoly$^{\mathsf{REG}}$	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

TO = Timeout of 300s, RE = Runtime Error

Figure 6 Experiments SMALL, MEDIUM, LARGE, and INFINITE.

satisfies $\mathsf{con}_{\mathsf{vgt}}(y, Q_y, \mathcal{G})$ for some set $\mathcal{G}$ such that $\{y\} \subsetneq \mathsf{fv}(Q_{qp})$ for every $Q_{qp} \in \mathcal{G}$, and further assumes that Q has no subquery of the form $x \approx \mathsf{c}$, no closed atomic predicate, and no repeated predicate symbols. We control the size of the structure $\mathcal{S}$ in our experiments using a parameter $n = |\mathcal{T}^+| = |\mathcal{T}^-|$. All our experiments also used a Data Golf structure with $n = 2$ as the training database for RC2SQL and VGT. We remark that relations in a Data Golf structure have typically more than n tuples because the sets $\mathcal{T}^+$, $\mathcal{T}^-$ grow in the recursion on subqueries. We use two strategies to obtain Data Golf structures (parameter γ in Appendix F).

In the SMALL, MEDIUM, and LARGE experiments, we generate ten pseudorandom queries with a fixed size 14. 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 2 free variables and every subquery has at most 4 free variables. The values of the parameter n for Data Golf structures are summarized in Figure 6.

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

Figure 6 shows the empirical evaluation results for the experiments SMALL, MEDIUM, LARGE, and INFINITE. All entries are execution times in seconds, TO is a timeout, and RE is a runtime error. Each column shows evaluation times for a unique pseudorandom query.

Query	Q^{susp}		Q^{susp}_{user}		Q^{susp}_{text}	
Param. n	10^3	10^4	10^3	10^4	10^3	10^4
RC2SQL	**2.0**	**2.4**	**3.0**	**3.5**	**6.3**	**7.4**
RC2SQL$^-$	62.2	TO	62.8	TO	397.2	TO
VGT	2.9	2.9	–	–	208.8	TO
VGT$^-$	TO	TO	–	–	TO	TO
DDD	6.3	TO	6.4	TO	28.9	TO
LDD	33.4	TO	36.3	TO	209.9	TO
MonPoly$^{\text{REG}}$	49.8	TO	47.3	TO	179.7	TO

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

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

Figure 7 Experiment with the queries Q^{susp}, Q^{susp}_{user}, and Q^{susp}_{text}.

The lowest time for a query is typeset in bold. We do not report the translation time because it does not contribute to the time complexity for a fixed query. Still, RC2SQL's translation time is below 0.6 seconds in all experiments. We also omit the rows for tools that time out or crash on all queries of an experiment, e.g., Ailamazyan et al. [2]. We conclude that our translation RC2SQL significantly outperforms all other tools on all queries and scales well to higher values of n, i.e., larger relations in the Data Golf structures, on all queries.

We also evaluate the tools on the queries Q^{susp} and Q^{susp}_{user} from introduction and on the more challenging query $Q^{susp}_{text} \coloneqq \mathsf{B}(b) \wedge \exists u, s, t.\ \forall p.\ \mathsf{P}(b, p) \longrightarrow \mathsf{S}(p, u, s) \vee \mathsf{T}(p, u, t)$ with an additional relation T that relates user's review text (variable t) to a product. The query Q^{susp}_{text} computes all brands for which there is a user, a score, and a review text such that all the brand's products were reviewed by that user with that score or by that user with that text. We use both pseudorandom Data Golf structures (strategy $\gamma = 1$) and real-world structures obtained from the Amazon review dataset [23]. The real-world relations P, S, and T are obtained by projecting the respective tables from the Amazon review dataset for some chosen product categories (abbreviated GC and MI in Figure 7) and the relation B contains all brands from P that have at least three products. Because the tool by Ailamazyan et al., DDD, LDD, and MonPoly$^{\text{REG}}$ only support integer data, we injectively remap the string and floating-point values from the Amazon review dataset to integers.

Figure 7 shows the empirical evaluation results: execution times on Data Golf structures (left) and execution times on structures derived from the real-world dataset for two specific product categories (right). We remark that VGT cannot handle the query Q^{susp}_{user} as it is not evaluable [14]. Our translation RC2SQL significantly outperforms all other tools (except VGT on Q^{susp}, but RC2SQL still outperforms VGT) on both pseudorandom and real-world structures. VGT$^-$ translates Q^{susp} into a RANF query with a higher query cost than RC2SQL$^-$. However, the optimization $\mathsf{optcnt}(\cdot)$ manages to rectify this inefficiency and thus VGT exhibits a comparable performance as RC2SQL. Specifically, the factor of 80× in query cost between VGT$^-$ and RC2SQL$^-$ improves to 1.1× in query cost between VGT and RC2SQL on a Data Golf structure with $n = 20$ [25]. Nevertheless, VGT does not finish evaluating the query Q^{susp}_{text} within 10 minutes, unlike RC2SQL.

6 Conclusion

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

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.
2. Alfred K. Ailamazyan, Mikhail M. Gilula, Alexei P. Stolboushkin, and Grigorii F. Schwartz. Reduction of a relational model with infinite domains to the case of finite domains. *Dokl. Akad. Nauk SSSR*, 286:308–311, 1986. URL: `http://mi.mathnet.ru/dan47310`.
3. Arnon Avron and Yoram Hirshfeld. On first order database query languages. In *LICS, July 15-18, 1991, Amsterdam, The Netherlands*, pages 226–231. IEEE Computer Society, 1991. `doi:10.1109/LICS.1991.151647`.
4. David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015. `doi:10.1145/2699444`.
5. Michael Benedikt and Leonid Libkin. Relational queries over interpreted structures. *J. ACM*, 47(4):644–680, 2000. `doi:10.1145/347476.347477`.
6. Achim Blumensath and Erich Grädel. Finite presentations of infinite structures: Automata and interpretations. *Theory Comput. Syst.*, 37(6):641–674, 2004. `doi:10.1007/s00224-004-1133-y`.
7. Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Decision diagrams for linear arithmetic. In *FMCAD, 15-18 November 2009, Austin, Texas, USA*, pages 53–60. IEEE, 2009. `doi:10.1109/FMCAD.2009.5351143`.
8. Jan Chomicki and David Toman. Implementing temporal integrity constraints using an active DBMS. *IEEE Trans. Knowl. Data Eng.*, 7(4):566–582, 1995. `doi:10.1109/69.404030`.
9. Jens Claußen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB, August 25-29, 1997, Athens, Greece*, pages 286–295. Morgan Kaufmann, 1997. URL: `http://www.vldb.org/conf/1997/P286.PDF`.
10. E. F. Codd. Relational completeness of data base sublanguages. *Research Report / RJ / IBM / San Jose, California*, RJ987, 1972.
11. Erling Ellingsen. Regex golf, 2013. `https://alf.nu/RegexGolf`.
12. Martha Escobar-Molano, Richard Hull, and Dean Jacobs. Safety and translation of calculus queries with scalar functions. In Catriel Beeri, editor, *PODS, May 25-28, 1993, Washington, DC, USA*, pages 253–264. ACM Press, 1993. `doi:10.1145/153850.153909`.
13. Allen Van Gelder and Rodney W. Topor. Safety and correct translation of relational calculus formulas. In Moshe Y. Vardi, editor, *PODS, March 23-25, 1987, San Diego, California, USA*, pages 313–327. ACM, 1987. `doi:10.1145/28659.28693`.
14. Allen Van Gelder and Rodney W. Topor. Safety and translation of relational calculus queries. *ACM Trans. Database Syst.*, 16(2):235–278, 1991. `doi:10.1145/114325.103712`.
15. Richard Hull and Jianwen Su. Domain independence and the relational calculus. *Acta Informatica*, 31(6):513–524, 1994. `doi:10.1007/BF01213204`.
16. Michael Kifer. On safety, domain independence, and capturability of database queries (preliminary report). In Catriel Beeri, Joachim W. Schmidt, and Umeshwar Dayal, editors, *Proceedings of the Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness, June 28-30, 1988, Jerusalem, Israel*, pages 405–415. Morgan Kaufmann, 1988. `doi:10.1016/b978-1-4832-1313-2.50037-8`.
17. Nils Klarlund and Anders Møller. *MONA v1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001. URL: `http://www.brics.dk/mona/`.
18. Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. URL: `http://www.cs.toronto.edu/%7Elibkin/fmt`, `doi:10.1007/978-3-662-07003-1`.
19. Hong-Cheu Liu, Jeffrey Xu Yu, and Weifa Liang. Safety, domain independence and translation of complex value database queries. *Inf. Sci.*, 178(12):2507–2533, 2008. `doi:10.1016/j.ins.2008.02.005`.

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

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

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

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

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

25 Martin Raszyk, David Basin, Srđan Krstić, and Dmitriy Traytel. Implementation and evaluation associated with this paper, 2021. https://github.com/rc2sql/rc2sql.

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

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

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

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

A Evaluable Queries

The classes of *evaluable* queries [14, Def. 5.2] and *allowed* queries [14, 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 [14, Theorem 10.5]. Every evaluable query can be translated to an equivalent allowed query [14, Theorem 8.6]. Every allowed query can be translated to an equivalent RANF query [14, Theorem 9.6].

▶ **Definition 16.** *A query Q is called* evaluable *if*

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

A query Q is called allowed *if*

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

where the relation $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ is defined to hold iff there exists a set $\mathcal{G}$ such that $\mathsf{gen}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ and the relation $\mathsf{con}_{\mathsf{vgt}}(x, Q)$ is defined to hold iff there exists a set $\mathcal{G}$ such that $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{G})$, respectively. The relations $\mathsf{gen}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ and $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ are defined in Figure 8.

In Figure 9 we introduce a measure $\mathsf{measure}(Q)$ on queries, that decreases for proper subqueries, after pushing negation, and after distributing existential quantification over disjunction. Hence, the termination of the rules in Figures 2, 3, and 8 and the termination of the functions in Figures 13 and 14 follow using the measure $\mathsf{measure}(Q)$.

We relate the definitions from Figure 2 and Figure 8 with the following lemmas.

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

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

Figure 8 The relations $\mathsf{gen}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ and $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ [14].

▶ **Lemma 17.** *Let x and y be free variables in a query Q such that $\mathsf{gen}_{\mathsf{vgt}}(x, \neg Q)$ and $\mathsf{gen}_{\mathsf{vgt}}(y, Q)$ hold. Then we get a contradiction.*

Proof. The lemma is proved by induction on the query Q using the measure $\mathsf{measure}(Q)$ on queries defined in Figure 9, which decreases in every case of the definition in Figure 8. ◀

▶ **Lemma 18.** *Let Q be a query such that $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$ holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q. Suppose that $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ holds for a free variable $x \in \mathsf{fv}(Q)$. Then $\mathsf{gen}(x, Q)$ holds.*

Proof. The lemma is proved by induction on the query Q using the measure $\mathsf{measure}(Q)$ on queries defined in Figure 9, which decreases in every case of the definition in Figure 8.

Lemma 17 and the assumption that $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$ holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q imply that $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ cannot be derived using the rule $\mathsf{gen}_{\mathsf{vgt}}(x, \neg\exists y.\, Q_y)$, i.e., Q cannot be of the form $\neg\exists y.\, Q_y$. Every other case in the definition of $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ has a corresponding case in the definition of $\mathsf{gen}(x, Q)$. ◀

▶ **Lemma 19.** *Let Q be an* allowed *query, i.e., $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ holds for every free variable $x \in \mathsf{fv}(Q)$ and $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$ holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q. Then Q is a safe-range query, i.e., $\mathsf{gen}(x, Q)$ holds for every free variable $x \in \mathsf{fv}(Q)$ and $\mathsf{gen}(y, Q_y)$ holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q.*

Proof. The lemma is proved by applying Lemma 18 to every free variable of Q and to the bound variable y in every subquery of Q of the form $\exists y.\, Q_y$. ◀

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

Figure 9 The measure $\mathsf{measure}(Q)$ on RC queries.

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

Figure 10 Constant propagation rules.

Lemma 19 shows that every allowed query is safe-range. But there exist safe-range queries that are not allowed, e.g., $\mathsf{B}(x) \wedge x \approx y$.

B Constant Propagation

We introduce constant propagation rules in Figure 10. We denote by $\mathsf{cp}(Q)$ the query obtained from a query Q by exhaustively applying the rules in Figure 10. Note that $\mathsf{cp}(Q)$ is either of the form $\bot$ or $\top$ or contains neither $\bot$ nor $\top$ as a subquery.

The following definitions introduce substitution of a variable by another variable and removing all free occurrences of a free variable.

▶ **Definition 20.** *The substitution of the form $Q[x \mapsto y]$ is the query $\mathsf{cp}(Q')$ where Q' is obtained from a query Q by replacing all occurrences of the free variable x by the variable y, potentially also renaming bound variables to avoid capture.*

▶ **Definition 21.** *The substitution of the form $Q[x/\bot]$ is the query $\mathsf{cp}(Q')$ where Q' is obtained from a query Q by replacing with $\bot$ every atomic predicate or equality containing the free variable x, except for $(x \approx x) \equiv \top$.*

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 admit a simple construction of an equivalent RA expression. 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 query rather than functional dependencies between attributes in relations (e.g., as in 1NF, 2NF, 3NF).

Figure 11 defines the predicate $\mathsf{ranf}(\cdot)$ characterizing RANF queries. The translation of safe-range queries (Section 3.2) to equivalent RANF queries proceeds via safe-range normal form (SRNF) introduced in [1, Section 5.4] and summarized here in Appendix C.1. A safe-range query in SRNF can be translated to an equivalent RANF query by subquery

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

Figure 11 Characterization of RANF queries.

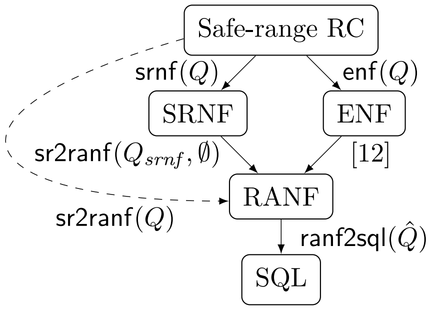

Figure 12 Overview of query normal forms.

rewriting using the following rules [1, Algorithm 5.4.7]:

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

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

Existential normal form (ENF) was introduced by Van Gelder and Topor [14] to translate an allowed query [14] into an equivalent RANF query. Given a safe-range query in ENF, the rules $(R1)$–$(R3)$ can be applied to obtain an equivalent RANF query [12, Lemma 7.8]. We remark that the rules $(R1)$–$(R3)$ are not sufficient to yield an equivalent RANF query for the original definition of ENF [14]. This issue has been identified and fixed by Escobar-Molano et al. [12]. Unlike SRNF, a query in ENF can have a subquery of the form $\neg(Q_1 \wedge Q_2)$, but no subquery of the form $\neg Q_1 \vee Q_2$ or $Q_1 \vee \neg Q_2$. A function $\mathsf{enf}(Q)$ that yields an ENF query equivalent to Q can be defined in terms of subquery rewriting using the rules in [12, Fig. 2]. Although applying the rules $(R1)$–$(R3)$ to $\mathsf{enf}(Q)$ instead of $\mathsf{srnf}(Q)$ may result in a RANF query with fewer subqueries, the query cost, i.e., the time complexity of query evaluation, can be arbitrarily larger. We illustrate this in the following example that is also included in our artifact [25]. We thus opt for using SRNF instead of ENF for translating safe-range queries into RANF.

▶ **Example 22.** The safe-range query $Q_{enf} := \mathsf{P}_2(x, y) \wedge \neg(\mathsf{P}_1(x) \wedge \mathsf{P}_1(y))$ is in ENF and RANF, but not SRNF. Applying the rule $(R1)$ to $\mathsf{srnf}(Q_{enf})$ yields the RANF query $Q_{srnf} :=$

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

function $\mathsf{srnf}(Q) =$
  switch Q **do**
    case $\neg Q'$ **do**
      switch Q' **do**
        case $\neg Q''$ **do return** $\mathsf{srnf}(Q'')$;
        case $Q_1 \vee Q_2$ **do return** $\mathsf{srnf}((\neg Q_1) \wedge (\neg Q_2))$;
        case $Q_1 \wedge Q_2$ **do return** $\mathsf{srnf}((\neg Q_1) \vee (\neg Q_2))$;
        case $\exists \vec{v}.\, Q_{\vec{v}}$ **do**
          if $\vec{v} \cap \mathsf{fv}(Q_{\vec{v}}) = \emptyset$ **then return** $\mathsf{srnf}(\neg Q_{\vec{v}})$;
          else
            switch $\mathsf{srnf}(Q_{\vec{v}})$ **do**
              case $Q_1 \vee Q_2$ **do return** $\mathsf{srnf}((\neg \exists \vec{v}.\, Q_1) \wedge (\neg \exists \vec{v}.\, Q_2))$;
              otherwise do return $\neg \exists \vec{v} \cap \mathsf{fv}(Q_{\vec{v}}).\, \mathsf{srnf}(Q_{\vec{v}})$;
        otherwise do return Q;
    case $Q_1 \vee Q_2$ **do return** $\mathsf{srnf}(Q_1) \vee \mathsf{srnf}(Q_2)$;
    case $Q_1 \wedge Q_2$ **do return** $\mathsf{srnf}(Q_1) \wedge \mathsf{srnf}(Q_2)$;
    case $\exists \vec{v}.\, Q_{\vec{v}}$ **do**
      switch $\mathsf{srnf}(Q_{\vec{v}})$ **do**
        case $Q_1 \vee Q_2$ **do return** $\mathsf{srnf}((\exists \vec{v}.\, Q_1) \vee (\exists \vec{v}.\, Q_2))$;
        otherwise do return $\exists \vec{v} \cap \mathsf{fv}(Q_{\vec{v}}).\, \mathsf{srnf}(Q_{\vec{v}})$;
    otherwise do return Q;

Figure 13 Translation to SRNF.

$(\mathsf{P}_2(x,y) \wedge \neg \mathsf{P}_1(x)) \vee (\mathsf{P}_2(x,y) \wedge \neg \mathsf{P}_1(y))$ that is equivalent to Q_{enf}. The costs of the two queries over a structure $\mathcal{S}$ are $\mathsf{cost}^{\mathcal{S}}(Q_{enf}) = 2 \cdot |[\![\mathsf{P}_2(x,y)]\!]| + |[\![\mathsf{P}_1(x)]\!]| + |[\![\mathsf{P}_1(y)]\!]| + 2 \cdot |[\![\mathsf{P}_1(x) \wedge \mathsf{P}_1(y)]\!]| + 2 \cdot |[\![Q_{enf}]\!]|$ and $\mathsf{cost}^{\mathcal{S}}(Q_{srnf}) = 2 \cdot |[\![\mathsf{P}_2(x,y)]\!]| + |[\![\mathsf{P}_1(x)]\!]| + 2 \cdot |[\![\mathsf{P}_2(x,y)]\!]| + |[\![\mathsf{P}_1(y)]\!]| + 2 \cdot |[\![\mathsf{P}_2(x,y) \wedge \neg \mathsf{P}_1(x)]\!]| + 2 \cdot |[\![\mathsf{P}_2(x,y) \wedge \neg \mathsf{P}_1(y)]\!]| + 2 \cdot |[\![Q_{srnf}]\!]|$, respectively. Note that the cost of Q_{enf} can be arbitrarily larger if $\mathsf{P}_1(x) \wedge \mathsf{P}_1(y)$ evaluates to a large intermediate result, i.e., $|[\![\mathsf{P}_1(x) \wedge \mathsf{P}_1(y)]\!]| \gg |[\![\mathsf{P}_2(x,y)]\!]|$. In contrast, the cost of Q_{srnf} can only be larger by a constant factor.

Figure 12 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 of a safe-range query to RANF we opt for in this paper. It is the composition of the two translations from safe-range RC to SRNF and from SRNF to RANF, respectively. In the rest of this section we introduce the normal forms and translations. Appendix E.3 shows how we translate RANF queries to SQL.

C.1 Safe-Range Normal Form

A query Q is in safe-range normal form (SRNF) if the query Q' in every subquery $\neg Q'$ of Q has the form of an atomic predicate, equality, or an existentially quantified query [1]. Figure 13 defines the function $\mathsf{srnf}(Q)$ that yields a SRNF query equivalent to a query Q. The function $\mathsf{srnf}(Q)$ proceeds by pushing negation [1, Section 5.4], distributing existential quantifiers over disjunction [14, Rule (T9)], and dropping bound variables that never occur [14, Definition 9.2]. We include the last two rules to optimize the time complexity of evaluating the resulting RANF query. The termination of the function $\mathsf{srnf}(Q)$ follows using the measure $\mathsf{measure}(Q)$, defined in Figure 9.

Using Figure 2, if a query Q is safe-range, then $\mathsf{srnf}(Q)$ is also safe-range. Next we prove the following lemma that we use as a precondition for translating safe-range SRNF queries to RANF queries.

▶ **Lemma 23.** *Let Q_{srnf} be a query in SRNF. Then $\mathsf{gen}(x, \neg Q')$ does not hold for any variable x and subquery $\neg Q'$ of $\neg Q_{srnf}$.*

Proof. Using Figure 2, $\mathsf{gen}(x, \neg Q')$ can only hold if $\neg Q'$ has the form $\neg\neg Q$, $\neg(Q_1 \vee Q_2)$, or $\neg(Q_1 \wedge Q_2)$. The SRNF query Q_{srnf} cannot have a subquery $\neg Q'$ that has any such form. ◀

C.2 Relational Algebra Normal Form

The function $\mathsf{sr2ranf}(Q, \mathcal{Q}) = (\hat{Q}, \overline{\mathcal{Q}})$, defined in Figure 14, where $\mathsf{sr2ranf}$ stands for *safe-range to relational algebra normal form*, takes a safe-range query $Q \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ in SRNF and returns a RANF query $\hat{Q}$ and a subset $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ such that $Q \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q} \equiv \hat{Q} \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$. To restrict variables in Q, the function $\mathsf{sr2ranf}(Q, \mathcal{Q})$ conjoins a subset of queries $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ to Q. Given a safe-range query Q, we first convert Q into SRNF and set $\mathcal{Q} = \emptyset$. Then we define $\mathsf{sr2ranf}(Q) := \hat{Q}$, where $(\hat{Q}, \_) := \mathsf{sr2ranf}(\mathsf{srnf}(Q), \emptyset)$, to be a RANF query $\hat{Q}$ equivalent to Q. The termination of $\mathsf{sr2ranf}(Q, \mathcal{Q})$ follows from the lexicographic measure $(2 \cdot \mathsf{measure}(Q) + \mathsf{eqneg}(Q) + 2 \cdot \sum_{\overline{Q} \in \mathcal{Q}} \mathsf{measure}(\overline{Q}) + 2 \cdot |\mathcal{Q}|, \mathsf{measure}(Q))$, where $\mathsf{measure}(Q)$ is defined in Figure 9, $\mathsf{eqneg}(Q) := 1$ if Q has the form of an equality between two variables or a negation, and $\mathsf{eqneg}(Q) := 0$ otherwise.

Next we describe the definition of $\mathsf{sr2ranf}(Q, \mathcal{Q})$ that follows [1, Algorithm 5.4.7]. Note that no constant propagation (Appendix B) is needed in [1, Algorithm 5.4.7], because the constants $\bot$ and $\top$ are not in the query syntax [1, Section 5.3]. Because $\mathsf{gen}(x, \bot)$ holds and $x \notin \mathsf{fv}(\bot)$, we need to perform constant propagation to guarantee that every disjunct has the same set of free variables (e.g., the query $\bot \vee \mathsf{B}(x)$ must be translated to $\bot$ to be in RANF). We flatten the disjunction and conjunction using $\mathsf{flat}^{\vee}(\cdot)$ and $\mathsf{flat}^{\wedge}(\cdot)$, respectively. In the case of a conjunction $Q^{\wedge}$, we first split the queries from $\mathsf{flat}^{\wedge}(Q^{\wedge})$ into queries $\mathcal{Q}^{+}$ that do not have the form of a negation and queries $\mathcal{Q}^{+}$ that do. Then we take out equalities between two variables and negations of equalities between two variables from the sets $\mathcal{Q}^{+}$ and $\mathcal{Q}^{-}$, respectively. To partition $Q^{\wedge}$ this way, we define the predicates $\mathsf{neg}(Q)$ and $\mathsf{eq}(Q)$ characterizing equalities between two variables and negations, respectively, i.e., $\mathsf{neg}(Q)$ is true iff Q has the form $\neg Q'$ and $\mathsf{eq}(Q)$ is true iff Q has the form $x \approx y$. Finally, the function $\mathsf{sort}^{\wedge}(\mathcal{Q})$ converts a set of queries into a RANF conjunction, defined in Figure 11, i.e., a left-associative conjunction in RANF. Note that the function $\mathsf{sort}^{\wedge}(\mathcal{Q})$ must place the queries $x \approx y$ so that either x or y is free in some preceding conjunct, e.g., $\mathsf{B}(x) \wedge x \approx y \wedge y \approx z$ is in RANF, but $\mathsf{B}(x) \wedge y \approx z \wedge x \approx y$ is not. In the case of an existentially quantified query $\exists \vec{v}.\, Q_{\vec{v}}$, we rename the variables $\vec{v}$ to avoid clash of the free variables in the set of queries $\mathcal{Q}$ with the bound variables $\vec{v}$.

Finally, the nondeterministic choices in the function $\mathsf{sr2ranf}(Q, \mathcal{Q})$ are resolved by minimizing the cost of the resulting RANF query with respect to a training database (Appendix E.1).

D Complexity Analysis

For an atomic predicate $Q_{ap} \in \mathsf{aps}(Q)$, let $\mathcal{B}_Q(Q_{ap})$ be the set of sequences of bound variables for all occurrences of Q_{ap} in Q. For example, let $Q_{ex} := ((\exists z.\, (\exists y, z.\, \mathsf{P}_3(x, y, z)) \wedge \mathsf{P}_2(y, z)) \wedge$

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

function $\mathsf{sr2ranf}(Q, \mathcal{Q}) =$
  if $\mathsf{ranf}(Q)$ **then return** $(\mathsf{cp}(Q), \emptyset)$;
  switch Q **do**
    case $x \approx y$ **do return** $\mathsf{sr2ranf}(x \approx y \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}, \emptyset)$;
    case $\neg Q'$ **do**
      $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q} \mid (\neg Q') \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range$\}$;
      if $\overline{\mathcal{Q}} = \emptyset$ **then**
        $(\hat{Q}', \_) := \mathsf{sr2ranf}(Q', \emptyset)$;
        return $(\mathsf{cp}(\neg \hat{Q}'), \emptyset)$;
      else return $\mathsf{sr2ranf}((\neg Q') \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}, \emptyset)$;
    case $Q_1 \vee Q_2$ **do**
      $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q} \mid \bigvee_{Q' \in \mathsf{flat}^{\vee}(Q)} (Q' \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q})$ is safe-range$\}$;
      foreach $Q' \in \mathsf{flat}^{\vee}(Q)$ **do** $(\hat{Q}', \_) := \mathsf{sr2ranf}(Q' \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}, \emptyset)$;
      return $(\mathsf{cp}(\bigvee_{Q' \in \mathsf{flat}^{\vee}(Q)} \hat{Q}'), \overline{\mathcal{Q}})$;
    case $Q_1 \wedge Q_2$ **do**
      $\mathcal{Q}^- := \{Q' \in \mathsf{flat}^{\wedge}(Q) \mid \mathsf{neg}(Q')\}$; $\mathcal{Q}^+ := \mathsf{flat}^{\wedge}(Q) \setminus \mathcal{Q}^-$;
      $\mathcal{Q}^= := \{Q' \in \mathcal{Q}^+ \mid \mathsf{eq}(Q')\}$; $\mathcal{Q}^+ := \mathcal{Q}^+ \setminus \mathcal{Q}^=$;
      $\mathcal{Q}^{\neq} := \{\neg Q' \in \mathcal{Q}^- \mid \mathsf{eq}(Q')\}$; $\mathcal{Q}^- := \mathcal{Q}^- \setminus \mathcal{Q}^{\neq}$;
      foreach $Q' \in \mathcal{Q}^+$ **do** $(\hat{Q}', \mathcal{Q}_{Q'}) := \mathsf{sr2ranf}(Q', \mathcal{Q} \cup ((\mathcal{Q}^+ \cup \mathcal{Q}^=) \setminus \{Q'\}))$;
      foreach $\neg Q' \in \mathcal{Q}^-$ **do** $(\hat{Q}', \_) := \mathsf{sr2ranf}(Q', \mathcal{Q} \cup (\mathcal{Q}^+ \cup \mathcal{Q}^=))$;
      $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q}^+ \mid \mathcal{Q}^+ \subseteq \bigcup_{Q' \in \overline{\mathcal{Q}}} (\mathcal{Q}_{Q'} \cup \{Q'\})\}$;
      return $(\mathsf{cp}(\mathsf{sort}^{\wedge}(\bigcup_{Q' \in \overline{\mathcal{Q}}} \hat{Q}' \cup \mathcal{Q}^= \cup \bigcup_{\neg Q' \in \mathcal{Q}^-} \neg \hat{Q}' \cup \mathcal{Q}^{\neq})), \bigcup_{Q' \in \overline{\mathcal{Q}}} (\mathcal{Q}_{Q'} \cap \mathcal{Q}))$;
    case $\exists \vec{v}.\, Q_{\vec{v}}$ **do**
      if $\mathsf{fv}(\mathcal{Q}) \cap \vec{v} \neq \emptyset$ **then** $\vec{w} \leftarrow \{\vec{w} \mid |\vec{w}| = |\vec{v}|$ and $((\mathsf{fv}(Q_{\vec{v}}) \setminus \vec{v}) \cup \mathsf{fv}(\mathcal{Q})) \cap \vec{w} = \emptyset\}$;
      else $\vec{w} := \vec{v}$;
      $Q_{\vec{w}} := Q_{\vec{v}}[\vec{v} \mapsto \vec{w}]$;
      $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q} \mid Q_{\vec{w}} \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range$\}$;
      $(\hat{Q}_{\vec{w}}, \_) := \mathsf{sr2ranf}(Q_{\vec{w}} \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}, \emptyset)$;
      return $(\mathsf{cp}(\exists \vec{w}.\, \hat{Q}_{\vec{w}}), \overline{\mathcal{Q}})$;
    otherwise do return $(\mathsf{cp}(Q), \emptyset)$;

Figure 14 Translation of safe-range SRNF to RANF.

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

A crucial property of our translation that is central for the complexity analysis (Theorem 12) is formalized in the following lemma.

▶ **Lemma 24.** *Let Q be an RC query with pairwise distinct (free and bound) variables and let $\mathsf{rw}(Q) = (\hat{Q}_{fin}, \hat{Q}_{inf})$. Let $\hat{Q} \in \{\hat{Q}_{fin}, \hat{Q}_{inf}\}$. Then $\mathsf{qps}(\hat{Q}) \subseteq \overline{\mathsf{qps}}(Q)$.*

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

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

Suppose that $Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ is a safe-range query in which no variable occurs both free and bound, no bound variables shadow each other, i.e., there are no subqueries $\exists x.\, Q_x \sqsubseteq Q'_x$ and $\exists x.\, Q'_x \sqsubseteq Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$, and every two subqueries $\exists x.\, Q_x \sqsubseteq Q_1$ and $\exists x.\, Q'_x \sqsubseteq Q_2$ such that $Q_1 \wedge Q_2 \sqsubseteq Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ have the property that $\exists x.\, Q_x$ or $\exists x.\, Q'_x$ is a quantified predicate. Then the free variables in $\bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ never clash with the bound variables in Q', i.e., Line 24 in Figure 14 is never executed. Next we observe that $\mathcal{B}_{\mathsf{sr2ranf}(Q', \mathcal{Q})}(Q_{ap}) \subseteq \mathcal{B}_{Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}}(Q_{ap})$ and then $\mathsf{qps}(\mathsf{sr2ranf}(Q', \mathcal{Q})) \subseteq \mathsf{qps}(Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q})$. Because Q_{fin}, Q_{inf} have the properties from the beginning of this paragraph and $\mathsf{qps}(\mathsf{srnf}(Q')) \subseteq \mathsf{qps}(Q')$, for every query Q', we get $\mathsf{qps}(\hat{Q}_{fin}) = \mathsf{qps}(\mathsf{sr2ranf}(Q_{fin})) \subseteq \mathsf{qps}(Q_{fin})$ and $\mathsf{qps}(\hat{Q}_{inf}) = \mathsf{qps}(\mathsf{sr2ranf}(Q_{inf})) \subseteq \mathsf{qps}(Q_{inf})$. ◀

Recall Example 13. The query $\exists u, p.\, \mathsf{S}(p, u, s)$ is in $\mathsf{qps}(Q_{vgt})$, but not in $\overline{\mathsf{qps}}(Q)$. Hence, $\mathsf{qps}(Q_{vgt}) \subseteq \overline{\mathsf{qps}}(Q)$, i.e., an analogue of Lemma 24 for Van Gelder and Topor's translation, does not hold.

Let a structure $\mathcal{S}$ be fixed. We observe that every tuple satisfying a RANF query $\hat{Q}$ belongs to the set of tuples satisfying the join over some minimal subset $\mathcal{Q}_{qps} \subseteq \mathsf{qps}(\hat{Q})$ of quantified predicates and also satisfying equalities duplicating some columns from $\mathcal{Q}_{qps}$. Note that $\{x \approx y \mid x \in V \wedge y \in V'\}$ denotes the set of all equalities $x \approx y$ between variables $x \in V$ and $y \in V'$.

▶ **Lemma 25.** *Let $\hat{Q}$ be a RANF query. Then $\llbracket \hat{Q} \rrbracket$ satisfies*

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

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

Now we derive a bound on $\left| \llbracket \hat{Q}' \rrbracket \right|$, for an arbitrary RANF subquery $\hat{Q}' \sqsubseteq \hat{Q}$, $\hat{Q} \in \{\hat{Q}_{fin}, \hat{Q}_{inf}\}$.

▶ **Lemma 26.** *Let Q be an RC query with pairwise distinct (free and bound) variables and let $\mathsf{rw}(Q) = (\hat{Q}_{fin}, \hat{Q}_{inf})$. Let $\hat{Q}' \sqsubseteq \hat{Q}$ be a RANF subquery of $\hat{Q} \in \{\hat{Q}_{fin}, \hat{Q}_{inf}\}$. Then*

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

Proof. Applying Lemma 25 to the RANF query $\hat{Q}'$ yields

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

We observe that $\left|\left[\!\left[\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp} \wedge \bigwedge_{Q^{=} \in \mathcal{Q}^{=}} Q^{=}\right]\!\right]\right| \leq \left|\left[\!\left[\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp}\right]\!\right]\right| \leq \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]|$ where the first inequality follows from the fact that equalities $Q^{=} \in \mathcal{Q}^{=}$ can only restrict a set of tuples and duplicate columns. Because $\hat{Q}'$ is a subquery of $\hat{Q}$, it follows that $\mathsf{qps}(\hat{Q}') \subseteq \mathsf{qps}(\hat{Q})$. Lemma 24 yields $\mathsf{qps}(\hat{Q}) \subseteq \overline{\mathsf{qps}}(Q)$. Hence, we derive $\mathsf{qps}(\hat{Q}') \subseteq \overline{\mathsf{qps}}(Q)$.

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

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

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

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

▶ **Lemma 27.** *Let Q be an RC query with pairwise distinct (free and bound) variables and let $\mathsf{rw}(Q) = (\hat{Q}_{fin}, \hat{Q}_{inf})$. Let $\hat{Q} \in \{\hat{Q}_{fin}, \hat{Q}_{inf}\}$. Then*

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

Proof. Recall that $\left|\mathsf{sub}(\hat{Q})\right|$ denotes the number of subqueries of the query $\hat{Q}$ and thus bounds the number of RANF subqueries $\hat{Q}'$ of the query $\hat{Q}$. For every subquery $\hat{Q}'$ of $\hat{Q}$, we first use the fact that $\left|\mathsf{fv}(\hat{Q}')\right| \leq \left|\mathsf{av}(\hat{Q})\right|$ to bound $\left|\left[\!\left[\hat{Q}'\right]\!\right]\right| \cdot \left|\mathsf{fv}(\hat{Q}')\right| \leq \left|\left[\!\left[\hat{Q}'\right]\!\right]\right| \cdot \left|\mathsf{av}(\hat{Q})\right|$. Then we use the estimation of $\left|\left[\!\left[\hat{Q}'\right]\!\right]\right|$ by Lemma 26. ◀

Finally, we prove Theorem 12.

Proof of Theorem 12. We derive Theorem 12 from Lemma 27 and the fact that the quantities $\left|\mathsf{sub}(\hat{Q})\right|$, $\left|\mathsf{av}(\hat{Q})\right|$, and $2^{\left|\mathsf{av}(\hat{Q})\right|^2}$ only depend on the query Q and thus they do not contribute to the asymptotic time complexity of capturing a fixed query Q. ◀

E Implementation Details

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

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

where

$$\begin{aligned} Q'_{fin} &\coloneqq \mathsf{ranf2sql}(\mathsf{optcnt}(Q_{fin})), \\ Q'_{inf} &\coloneqq \mathsf{ranf2sql}(\mathsf{optcnt}(Q_{inf})), \\ (Q_{fin}, Q_{inf}) &\coloneqq \mathsf{rw}(Q). \end{aligned}$$

Function $\mathsf{rw}(\cdot)$ is defined in Section 4.4 as a composition of the $\mathsf{split}(\cdot)$ and $\mathsf{sr2ranf}(\cdot)$ functions, which are defined in Section 4.3 and Appendix C, respectively. Below we first describe how we resolve the nondeterministic choices in all our algorithms. Then we define functions $\mathsf{optcnt}(\cdot)$ and $\mathsf{ranf2sql}(\cdot)$.

E.1 Instantiating Our Translation

To resolve the nondeterministic choices in our algorithms, we suppose that the algorithms have access to a *training database* $\mathcal{T}$ of constant size. The training database is used to compare the cost of queries over the actual database and thus it should preserve the relative ordering of queries by their cost over the actual database as much as possible. Nevertheless, our translation satisfies the correctness and worst-case complexity claims (Section 4.3 and 4.4) for every choice of the training database. The training databases used in our empirical evaluation are obtained using the function dg (Appendix F) with parameters described in Section 5. Because of its constant size, the complexity of evaluating a query over the training database is constant and does not impact the asymptotic time complexity of evaluating the query over the actual database using our translation. There are two types of nondeterministic choices in our algorithms:

- Choosing some $X \in \mathcal{X}$ in a while-loop. As the while-loops always update $\mathcal{X}$ with $\mathcal{X} := (\mathcal{X} \setminus \{X\}) \cup f(X)$ for some f, the order in which the elements of $\mathcal{X}$ are chosen does not matter.
- Choosing a variable $x \in V$ and a set $\mathcal{G}$ such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$, where $\tilde{Q}$ is a query with range-restricted bound variables and $V \subseteq \mathsf{fv}(\tilde{Q})$ is a subset of its free variables. Observe that the measure $\mathsf{measure}(Q)$ on queries, defined in Figure 9, decreases for the queries in the premises of the rules for $\mathsf{gen}(x, \tilde{Q}, \mathcal{G})$ and $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$, defined in Figure 2 and 3. Hence, deriving $\mathsf{gen}(x, \tilde{Q}, \mathcal{G})$ and $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ either succeeds or gets stuck after at most $\mathsf{measure}(\tilde{Q})$ steps. In particular, we can enumerate all sets $\mathcal{G}$ such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ holds. Because we derive one additional query $\tilde{Q}[x \mapsto y]$ for every $y \in \mathsf{eqs}(x, \mathcal{G})$ and a single query $\tilde{Q} \wedge \mathsf{qps}^{\vee}(\mathcal{G})$, we choose $x \in V$ and $\mathcal{G}$ minimizing $|\mathsf{eqs}(x, \mathcal{G})|$ as the first objective and $\sum_{Q_{qp} \in \mathsf{qps}(\mathcal{G})} \mathsf{cost}^{\mathcal{T}}(Q_{qp})$ as the second objective. Our particular choice of $\mathcal{G}$ with $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ is merely a heuristic and does not provide any additional guarantees compared to every other choice of $\mathcal{G}$ with $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$.

E.2 Optimization using Count Aggregations

In this section, we introduce count aggregations and describe a generalization of Claußen et al. [9]'s approach to evaluate RANF queries using count aggregations. Consider the query

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

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

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

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

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

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

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

▶ **Lemma 28.** *Let $\exists \vec{v}.\, Q_{\vec{v}} \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \neg \overline{Q}$, $\mathcal{Q} \neq \emptyset$, be a RANF query. Let c, c' be fresh variables that do not occur in $\mathsf{fv}(Q_{\vec{v}})$. Then*

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

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

▶ **Lemma 29.** *Let $\hat{Q} \wedge \neg \exists \vec{v}.\, Q_{\vec{v}} \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \neg \overline{Q}$, $\mathcal{Q} \neq \emptyset$, be a RANF query. Let c, c' be fresh variables that do not occur in $\mathsf{fv}(\hat{Q}) \cup \mathsf{fv}(Q_{\vec{v}})$. Then*

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

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

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

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

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

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

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

Here c_1, c_2 are fresh variables that do not occur in $\mathsf{fv}(Q_{\vec{v}}) \cup \mathsf{fv}(Q'_{\vec{v}}) \cup \{c\}$. Note that mini-scoping is only a heuristic and it can both improve and harm the time complexity of query evaluation. We implement the translations from Lemmas 28 and 29 and mini-scoping in a function called $\mathsf{optcnt}(\cdot)$. Given a RANF query $\hat{Q}$, $\mathsf{optcnt}(\hat{Q})$ is an equivalent RANF query after introducing count aggregations and performing mini-scoping.

▶ **Example 30.** We show how we introduce count aggregations into the RANF query

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

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

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

E.3 Translating RANF to SQL

Our translation of a RANF query into SQL has two steps: we first translate the query to an equivalent RA expression, which we then translate to SQL using a publicly available RA interpreter radb [29].

We define the function $\mathsf{ranf2ra}(\hat{Q})$ translating RANF queries $\hat{Q}$ into equivalent RA expressions $\mathsf{ranf2ra}(\hat{Q})$. The translation is based on Algorithm 5.4.8 by Abiteboul et al. [1], which we modify as follows. We adjust the way closed RC queries are handled. Chomicki and Toman [8] observed that closed RC queries cannot be handled by SQL, since SQL allows neither empty projections nor 0-ary relations. They propose to use a unary auxiliary predicate $\mathsf{A} \in \mathcal{R}$ whose interpretation $\mathsf{A}^{\mathcal{S}} = \{\mathsf{t}\}$ always contains exactly one tuple t. Every closed query $\exists x.\, Q_x$ is then translated into $\exists x.\, \mathsf{A}(t) \wedge Q_x$ with an auxiliary free variable t. Every other closed query $\hat{Q}$ is translated into $\mathsf{A}(t) \wedge \hat{Q}$, e.g., $\mathsf{B}(42)$ is translated into $\mathsf{A}(t) \wedge \mathsf{B}(42)$. We also use the auxiliary predicate A to translate queries of the form $x \approx \mathsf{c}$ and $\mathsf{c} \approx x$ because the single tuple (t) in $\mathsf{A}^{\mathcal{S}}$ can be mapped to any constant c. Finally, we extend [1, Algorithm 5.4.8] with queries of the form $[\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c)$.

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

A RANF query $Q_1 \wedge \neg Q_2$, where $\mathsf{ranf}(Q_1)$, $\mathsf{ranf}(Q_2)$, and $\mathsf{fv}(Q_2) \subsetneq \mathsf{fv}(Q_1)$ is translated into RA expression $\mathsf{ranf2ra}(Q_1) \triangleright \mathsf{ranf2ra}(Q_2)$, where $\triangleright$ denotes the anti-join operator (also refered to as the *generalized* difference operator [1]) and $\mathsf{ranf2ra}(Q_1)$, $\mathsf{ranf2ra}(Q_2)$ are the equivalent relational algebra expressions for Q_1, Q_2, respectively. The radb interpreter only supports the anti-join operator $\mathsf{ranf2ra}(Q_1) \triangleright \mathsf{ranf2ra}(Q_2)$ expressed as $\mathsf{ranf2ra}(Q_1) - (\mathsf{ranf2ra}(Q_1) \bowtie \mathsf{ranf2ra}(Q_2))$, where $-$ denotes the set difference operator and $\bowtie$ denotes the natural join. Alternatively, the anti-join operator can be directly mapped to `LEFT JOIN` in SQL. We generalize radb to use `LEFT JOIN` since it performs better in our empirical evaluation [25].

The radb interpreter introduces a separate SQL subquery in a `WITH` clause for every subexpression in the RA expression. We extend radb to additionally perform common subquery elimination, i.e., to merge syntactically equal subqueries. Common subquery elimination is also assumed in our query cost (Section 3.3).

Finally, the function $\mathsf{ranf2sql}(\hat{Q})$ (Figure 12) is defined as $\mathsf{ranf2sql}(\hat{Q}) \coloneqq \mathsf{ra2sql}(\mathsf{ranf2ra}(\hat{Q}))$, i.e., as a composition of the two translations from RANF to RA and from RA to SQL.

input : A RC query Q with pairwise distinct (free and bound) variables satisfying CON, CST, VAR, REP, a sequence of distinct variables $\vec{v}$, $\mathsf{fv}(Q) \subseteq \vec{v}$, sets of tuples $\mathcal{T}^+_{\vec{v}}$, $\mathcal{T}^-_{\vec{v}}$ over $\vec{v}$ such that $\mathcal{T}^+_{\vec{v}}[x] \cap \mathcal{T}^-_{\vec{v}}[x] = \emptyset$, $\left|\mathcal{T}^+_{\vec{v}}[x]\right| \geq \left|\mathcal{T}^+_{\vec{v}}\right|$, and $\left|\mathcal{T}^-_{\vec{v}}[x]\right| \geq \left|\mathcal{T}^-_{\vec{v}}\right|$, for every $x \in \vec{v}$, a parameter $\gamma \in \{0, 1\}$.

output : A structure $\mathcal{S}$ such that $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq [\![Q]\!]$, $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \emptyset$, and $|[\![Q']\!]|$, $|[\![\neg Q']\!]|$ contain at least $\min\{\left|\mathcal{T}^+_{\vec{v}}\right|, \left|\mathcal{T}^-_{\vec{v}}\right|\}$ tuples each, for every $Q' \sqsubseteq Q$.

function $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma) =$
- **switch** Q **do**
  - **case** $r(t_1, \ldots, t_{\iota(r)})$ **do return** $\{r^{\mathcal{S}} \mapsto \mathcal{T}^+_{\vec{v}}[t_1, \ldots, t_{\iota(r)}]\}$;
  - **case** $x \approx y$ **do**
    - **if** *there exist* d, d' *such that* $d \neq d'$ *and* $(d, d') \in \mathcal{T}^+_{\vec{v}}[x, y]$, *or* $d = d'$ *and* $(d, d') \in \mathcal{T}^-_{\vec{v}}[x, y]$ **then**
      - fail
  - **case** $\neg Q'$ **do return** $\mathsf{dg}(Q', \vec{v}, \mathcal{T}^-_{\vec{v}}, \mathcal{T}^+_{\vec{v}}, \gamma)$;
  - **case** $Q_1 \vee Q_2$ *or* $Q_1 \wedge Q_2$ **do**
    - $(\mathcal{T}^1_{\vec{v}}, \mathcal{T}^2_{\vec{v}}) \leftarrow \{(\mathcal{T}^1_{\vec{v}}, \mathcal{T}^2_{\vec{v}}) \mid \left|\mathcal{T}^1_{\vec{v}}\right| = \left|\mathcal{T}^2_{\vec{v}}\right| = \min\{\left|\mathcal{T}^+_{\vec{v}}\right|, \left|\mathcal{T}^-_{\vec{v}}\right|\}$ and $\mathcal{T}^1_{\vec{v}}[x] \cap \mathcal{T}^2_{\vec{v}}[x] = \emptyset$ and $(\mathcal{T}^1_{\vec{v}}[x] \cup \mathcal{T}^2_{\vec{v}}[x]) \cap (\mathcal{T}^+_{\vec{v}}[x] \cup \mathcal{T}^-_{\vec{v}}[x]) = \emptyset$, for all $x \in \vec{v}\}$;
    - **if** $\gamma = 0$ **then**
      - **return** $\mathsf{dg}(Q_1, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \gamma) \cup \mathsf{dg}(Q_2, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}}, \gamma)$
    - **else**
      - **switch** Q **do**
        - **case** $Q_1 \vee Q_2$ **do**
          - **return** $\mathsf{dg}(Q_1, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \gamma) \cup \mathsf{dg}(Q_2, \vec{v}, \mathcal{T}^1_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^+_{\vec{v}}, \gamma)$
        - **case** $Q_1 \wedge Q_2$ **do**
          - **return** $\mathsf{dg}(Q_1, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^-_{\vec{v}}, \mathcal{T}^1_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \gamma) \cup \mathsf{dg}(Q_2, \vec{v}, \mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}}, \mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}}, \gamma)$
  - **case** $\exists y.\, Q_y$ **do**
    - $(\mathcal{T}^1_{\vec{v}\cdot y}, \mathcal{T}^2_{\vec{v}\cdot y}) \leftarrow \{(\mathcal{T}^1_{\vec{v}\cdot y}, \mathcal{T}^2_{\vec{v}\cdot y}) \mid \mathcal{T}^1_{\vec{v}\cdot y}[\vec{v}] = \mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^2_{\vec{v}\cdot y}[\vec{v}] = \mathcal{T}^-_{\vec{v}}$ and $\mathcal{T}^1_{\vec{v}}[y] \cap \mathcal{T}^2_{\vec{v}}[y] = \emptyset$ and $(\mathcal{T}^1_{\vec{v}}[y] \cup \mathcal{T}^2_{\vec{v}}[y]) \cap (\mathcal{T}^+_{\vec{v}}[y] \cup \mathcal{T}^-_{\vec{v}}[y]) = \emptyset\}$;
    - **return** $\mathsf{dg}(Q_y, \vec{v} \cdot y, \mathcal{T}^1_{\vec{v}\cdot y}, \mathcal{T}^2_{\vec{v}\cdot y}, \gamma)$

Figure 15 Computing the Data Golf structure.

F Data Golf Benchmark

We devise the *Data Golf* benchmark for generating synthetic databases. Given a query Q, the generated database guarantees that no subquery Q' of 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 less meaningful. We first make the following assumptions on Q:

- CON the bound variable y in every subquery $\exists y.\, Q_y$ of Q satisfies $\mathsf{con}_{\mathsf{vgt}}(y, Q_y, \mathcal{G})$ (Figure 8) for some set $\mathcal{G}$ such that $\{y\} \subsetneq \mathsf{fv}(Q_{qp})$ for every $Q_{qp} \in \mathcal{G}$, to avoid subqueries like $\exists y.\, \neg \mathsf{P}_2(x, y)$ and $\exists y.\, (\mathsf{P}_2(x, y) \vee \mathsf{P}_1(y))$;
- CST Q contains no subquery of the form $x \approx \mathsf{c}$ because such a subquery is satisfied by exactly one tuple;
- VAR Q contains no closed atomic predicate, e.g., $\mathsf{P}_1(42)$, because a closed subquery is either satisfied by all possible tuples or no tuple at all; and
- REP Q contains no repeated predicate symbols, to avoid subqueries like $\mathsf{P}_1(x) \wedge \neg \mathsf{P}_1(x)$.

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

Data Golf is loosely inspired by *regex golf* [11], 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 $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$ (Figure 15) that computes a structure $\mathcal{S}$ such that $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq [\![Q]\!]$, $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \emptyset$, and $|[\![Q']\!]|$, $|[\![\neg Q']\!]|$ contain at least $\min\{|\mathcal{T}^+_{\vec{v}}|, |\mathcal{T}^-_{\vec{v}}|\}$ tuples each, for every subquery Q' of a query Q, where $\vec{v}$ is a sequence of distinct variables such that $\mathsf{fv}(Q) \subseteq \vec{v}$, $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$ are sets of tuples over $\vec{v}$, and $\gamma \in \{0, 1\}$ is a *strategy*.

The function $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$ can fail on an equality between two variables $x \approx y$. In this case, the function $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$ does not compute any Data Golf structure. We define the *not-depth* of a subquery $x \approx y$ in Q as the number of subqueries that have the form of a negation among the queries $x \approx y \sqsubseteq \cdots \sqsubseteq Q$, i.e., the number of negations on the path between the subquery $x \approx y$ and Q's main constructor. To prevent failure, we generate the sets $\mathcal{T}^+_{\vec{v}}$, $\mathcal{T}^-_{\vec{v}}$ to only contain tuples with equal values for all variables in equalities with even (odd, respectively) not-depth and pairwise distinct values for all variables in equalities with odd (even, respectively) not-depth. This is not always possible, e.g., for $x \approx y \wedge \neg x \approx y$, in which case no Data Golf structure can be computed.

In the case of a conjunction or a disjunction, we add disjoint sets $\mathcal{T}^1_{\vec{v}}$, $\mathcal{T}^2_{\vec{v}}$ of tuples over $\vec{v}$ to $\mathcal{T}^+_{\vec{v}}$, $\mathcal{T}^-_{\vec{v}}$ so that the intermediate results for the subqueries are neither equal nor disjoint.

Finally, we justify why a Data Golf structure $\mathcal{S}$ computed by $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$ satisfies $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq [\![Q]\!]$ and $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \emptyset$. We proceed by induction on the query Q. Because of REP, the Data Golf structures for the subqueries Q_1, Q_2 of a binary query $Q_1 \vee Q_2$, $Q_1 \wedge Q_2$ can be combined using the union operator. The only case that does not follow immediately is that $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \emptyset$ for a query Q of the form $\exists y.\, Q_y$. We prove this case by contradiction. Without loss of generality we assume that $\vec{\mathsf{fv}}(Q_y) = \vec{\mathsf{fv}}(Q) \cdot y$. Suppose that $\vec{d} \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)]$ and $\vec{d} \in [\![Q]\!]$. Because $\vec{d} \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)]$, there exists some d such that $\vec{d} \cdot d \in \mathcal{T}^2_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$. Because $\vec{d} \in [\![Q]\!]$, there exists some d' such that $\vec{d} \cdot d' \in [\![Q_y]\!]$. By the induction hypothesis, we get $\vec{d} \cdot d \notin [\![Q_y]\!]$ and $\vec{d} \cdot d' \notin \mathcal{T}^2_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$. Because $\mathsf{con}_{\mathsf{vgt}}(y, Q_y, \mathcal{G})$ holds for some $\mathcal{G}$ satisfying the assumption of CON, the query Q_y is equivalent to

$$Q_y \equiv (Q_y \wedge \mathsf{qps}^{\vee}(\mathcal{G})) \vee \bigvee_{z \in \mathsf{eqs}(y, \mathcal{G})} (Q_y[y \mapsto z]) \vee Q_y[y/\bot].$$

Recall that $\vec{d} \cdot d' \in [\![Q_y]\!]$. If the tuple $\vec{d} \cdot d'$ satisfies $Q_y[y \mapsto z]$ for some $z \in \mathsf{eqs}(y, \mathcal{G})$ or $Q_y[y/\bot]$, then $\vec{d} \cdot d \in [\![Q_y]\!]$ (contradiction) because the variable y does not occur in the queries $Q_y[y \mapsto z]$ and $Q_y[y/\bot]$ and thus its valuation in $\vec{d} \cdot d'$ can be arbitrarily changed. Otherwise, the tuple $\vec{d} \cdot d'$ satisfies some quantified predicate $Q_{qp} \in \mathsf{qps}(\mathcal{G})$ such that $\{y\} \subsetneq \mathsf{fv}(Q_{qp})$ (CON). Hence, the tuples $\vec{d} \cdot d$ and $\vec{d} \cdot d'$ agree on the valuation of a variable $x \in \mathsf{fv}(Q_{qp}) \setminus \{y\}$. Next we observe that the valuations of every variable (e.g., x) in the tuples of the sets $\mathcal{T}^+_{\vec{v}}$, $\mathcal{T}^-_{\vec{v}}$ are pairwise distinct (the conditions $\mathcal{T}^+_{\vec{v}}[x] \cap \mathcal{T}^-_{\vec{v}}[x] = \emptyset$, $|\mathcal{T}^+_{\vec{v}}[x]| \geq |\mathcal{T}^+_{\vec{v}}|$, and $|\mathcal{T}^-_{\vec{v}}[x]| \geq |\mathcal{T}^-_{\vec{v}}|$). Because $\vec{d} \cdot d \in \mathcal{T}^2_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$, $\vec{d} \cdot d'$ satisfies the quantified predicate Q_{qp} with $y \in \mathsf{fv}(Q_{qp})$, and the tuples $\vec{d} \cdot d$ and $\vec{d} \cdot d'$ agree on the valuation of x, the valuations of y in these two tuples must be equal, i.e., $d = d'$ (contradiction).

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