String Operations in Query Languages[*]

Michael Benedikt[†] Bell Labs

Leonid Libkin[§] U. Toronto

Thomas Schwentick[¶] U. Mainz

Luc Segoufin[‖] INRIA

Abstract

We study relational calculi with support for string operations. While SQL restricts the ability to mix string pattern-matching and relational operations, prior proposals for embedding SQL in a compositional calculus were based on adding the operation of concatenation to first-order logic. These latter proposals yield compositional query languages extending SQL, but are unfortunately computationally complete. The unbounded expressive power in turn implies strong limits on the ability to perform optimization and static analysis of properties such as query safety in these languages.

In contrast, we look at compositional extensions of relational calculus that have nice expressiveness, decidability, and safety properties, while capturing string-matching queries used in SQL. We start with an extension based on the string ordering and `LIKE` predicates. This extension shares some of the attractive properties of relational calculus (e.g. effective syntax for safe queries, low data complexity), but lacks the full power of regular-expression pattern-matching. When we extend this basic model to include string length comparison, we get a natural string language with great expressiveness, but one which includes queries with high (albeit bounded) data complexity. We thus explore the space between these two languages. We consider two intermediate languages: the first extends our base language with functions that trim/add leading characters, and the other extends it by adding the full power of regular-expression pattern-matching. We show that both these extensions inherit many of the attractive properties of the basic model: they both have corresponding algebras expressing safe queries, and low complexity of query evaluation.

1 Introduction

String manipulation facilities have long been recognized as a critical component of a realistic database query language. In SQL, for example, the `WHERE` clause can contain string *pattern-matching expressions*, such as `FACULTY.NAME LIKE 'Nyk%nen'`. These expressions can themselves be seen as queries over string relations: the above clause, for example, can be seen as a selection performed on a projection of the `FACULTY` relation. While the Relational Calculus gives a satisfactory formal model for SQL queries in the absence of built-in datatypes, there has been thus far no satisfactory model that fully accounts for string queries. The lack of an adequate formal model is related to the fact that SQL restricts the interaction of string operations and relational operations in a number of ad-hoc ways: one cannot apply the `LIKE` operator to a subquery to build up a new query, nor can one take the product of two string expressions built with `LIKE`. Our goal here is to present query languages that extend relational calculus to include string pattern-matching in such a way that string queries can be freely composed with relational expressions. The resulting languages can be used as formal tools for analyzing the complexity and expressiveness of SQL, and could also serve as target query languages amenable to the optimizations that result from the interaction of relational and string operations. In this pa-

---

[*] Part of this work was done while the second and the third authors visited INRIA, and the second and the fourth authors visited Mainz.

[†] Bell Laboratories, 263 Shuman Blvd, Naperville, IL 60566, USA. E-mail: benedikt@research.bell-labs.com.

[§] Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Ontario M5S 3H5, Canada. E-mail: libkin@cs.toronto.edu. Research affiliation: Bell Labs.

[¶] Johannes Gutenberg-Universität Mainz, Institut für Informatik, 55099 Mainz, Germany. Email: tick@informatik.uni-mainz.de.

[‖] INRIA-Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France. E-mail: Luc.Segoufin@inria.fr.

per we explore several such languages, and study their expressive power, complexity, and static analysis properties.

Some approaches toward unifying string algebras with relational algebra have been developed in the prior literature. [17] studied the consequences of adding pattern-matching features to SQL. [19, 22, 18] proposed an extension of the relational calculus with alignment logics and studied their complexity and expressive power, while [11, 12] considered Datalog extended with appropriate transducers for string operations, proving a number of completeness results. In [14] arbitrary regions (substrings) can be queried; this, when coupled with relational calculus, gives the power of string concatenation. Closer to our approach, [20, 26] study the relational calculus/algebra extended with an operation for concatenating strings. [15] studies first-order logic over term algebras and extends expressive bounds and complexity results from relational calculus to this setting. But SQL-style string pattern-matching cannot be expressed in the language of [15] – indeed in this language one cannot even query for strings beginning with a fixed symbol.

One problem faced in any work combining string pattern-matching queries with relational calculus is that pattern-matching expressions may return an infinite number of strings. This is the standard issue of *safety.* The authors tackle this problem by identifying safe fragments of their languages, using a number of syntactic restrictions — see, e.g., [19, 22, 18, 20, 26] — but they cannot capture the safe fragment of the language syntactically. A second problem concerns expressive power. Many query languages designed in the prior literature turn out to be Turing complete, a feature that in turn makes many sorts of analysis and optimization impossible. Indeed, as noted in [20], adding just concatenation to the relational calculus already yields a query language which is Turing complete. This immediately implies that there is no effective syntax for the corresponding safe fragment [28].

In contrast to the above, in our work we seek languages that fulfill the following criteria:

1. Query evaluation is efficient;
2. There is effective syntax capturing safe queries;
3. There is an algebra equivalent to the language.

We start with the observation that `LIKE` expressions correspond to first-order formulae over the model $\mathbf{S}$ consisting of strings with the prefix operation and functions for concatenating alphabet symbols. We thus consider relational calculus, RC, over this model: the resulting query language captures basic SQL with simple `LIKE` pattern-matching and lexicographic ordering. We show that the safe fragment of this model can be effectively captured in a natural way, and prove complexity bounds for queries in this language that match the known bounds for ordinary relational calculus. $\mathrm{RC}(\mathbf{S})$ however, is unable to express certain natural queries, e.g., `SELECT` $a \cdot x$ `FROM` R, where a is a fixed character. We thus extend this language to $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$ by introducing string length, more precisely, string length comparisons. This extension has much greater expressiveness: it enables additional operations such as trimming/adding symbols on both left and right of a string, the `SIMILAR` pattern-matching for checking membership in a regular language [21], and pattern-matching on sets of n-tuples. We show that this language also satisfies criteria 2 and 3 above, but in $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$ one can express NP-complete and coNP-complete problems.

This leads us to the consideration of two intermediate languages, $\mathrm{RC}(\mathbf{S}_{\mathrm{left}})$ and $\mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$. The first one adds operations for trimming/adding leading characters (that is, characters on the *left*), and the second one gives us regular expression pattern matching. Both languages satisfy all three of the required criteria, while considerably extending the expressive power of $\mathrm{RC}(\mathbf{S})$.

The paper is organized as follows. The next section presents the notations. Section 3 briefly reviews results on relational calculus with concatenation. Section 4 presents the sets of string operations considered in the paper. In Section 5 we explore the expressive power and complexity of $\mathrm{RC}(\mathbf{S})$ and $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$. Query safety for these languages is investigated in Section 6. In Section 7, we propose and study $\mathrm{RC}(\mathbf{S}_{\mathrm{left}})$ and $\mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$ and extend the previous results to these languages. Complete proofs of all results can be found in [9].

2 Notations

Strings and operations on them For a finite alphabet Σ, we write Σ^* for the set of all finite strings over Σ, and $\Sigma^{\leq n}$ for the set of all strings of length at most n. The empty string is denoted by ϵ. We shall consider a number of operations on strings; those used most often are:

- $x \cdot y$ is the concatenation of two strings x and y.
- $x \preceq y$ is true iff x is a prefix of y. $x \prec y$ is true iff x is a strict prefix of y.
- $l_a(x)$, $a \in \Sigma$, is a function that adds a as the *l*ast symbol: $l_a(x) = x \cdot a$.
- $f_a(x)$, $a \in \Sigma$, is a function that adds a as the *f*irst symbol: $f_a(x) = a \cdot x$.

- $|x|$ is the length of string x.
- $x - y$ is defined to be the relative suffix of y in x. That is, if $x = y \cdot z$, then $x - y = z$; otherwise $x - y = \epsilon$.
- $x \sqcap y$ is the longest common prefix of x and y.

We shall consider a number of first-order structures $\mathcal{M} = \langle \Sigma^*, \Omega \rangle$, where Ω is a collection of predicates and functions on Σ^*. Often it is convenient to have all relational symbols in Ω. For that purpose, we introduce the unary relation L_a (last symbol) which is true of x iff the last symbol of x is a, and a binary relation $F_a(x, y)$ which holds iff $y = f_a(x) = a \cdot x$.

Note that $|x|$ does not return a string, so it is not an operation of Σ^*. Instead, we use the binary predicate $\mathrm{el}(x, y)$ (equal length) which is true iff $|x| = |y|$.

We write $x \lessdot y$ to express that y extends x by exactly one symbol. Let *prefix*(C) stand for the *prefix-closure* of C: $\{s \mid s \preceq s', s' \in C\}$. By $\downarrow(C)$ we denote $\{s \mid |s| \leq |s'|, s' \in C\}$. Given $C \subseteq \Sigma^*$ and $x \in \Sigma^*$, by $x \sqcap C$ we denote the longest string among $x \sqcap c, c \in C$. Note that this is well-defined, since all the strings $x \sqcap c$ are prefixes of x.

Databases and query languages A database schema SC is a collection of relation names $R_1, \ldots, R_l$, R_i being of arity $p_i > 0$. In an instance of SC over a set U, each R_i is interpreted as a finite subset of U^{p_i}. The *active domain* of a database D, *adom*(D), is the set of elements from U that appear in D.

The general setting for query languages is that of a finite database and an infinite underlying structure $\mathcal{M} = \langle U, \Omega \rangle$, where Ω is a set of operations (functions and predicates) on U. As our basic language we consider relational calculus, or first-order logic, over the schema SC and $\mathcal{M}$, denoted by $\mathrm{RC}(\mathrm{SC}, \mathcal{M})$. We often omit SC when it is understood, or irrelevant. Here we will focus exclusively on the string datatype, hence we will always have $U = \Sigma^*$. For example, if $\mathcal{M} = \langle \Sigma^*, \prec, (L_a)_{a \in \Sigma} \rangle$, the query

$$\exists x\ R(x) \wedge L_0(x) \wedge \exists y (y \prec x \wedge L_1(y) \wedge (\neg \exists z\ y \prec z \prec x))$$

tests if there is a string in the relation R which ends with 10. Indeed, it asks if the last symbol of x is 0, and if there exists a prefix y, which is the largest prefix of x (as there is no z with $y \prec z \prec x$) such that the last symbol of y is 1.

Given a query $\varphi(x_1, \ldots, x_n)$ in $\mathrm{RC}(\mathrm{SC}, \mathcal{M})$ and $\vec{a} \in U^n$, we write $D \models \varphi(\vec{a})$ when $\varphi(\vec{a})$ is true in $(D, \mathcal{M})$. We write $\varphi(D)$ for the output of φ on D, that is, $\{\vec{a} \in U^n \mid D \models \varphi(\vec{a})\}$. We say that φ is *safe on* D if $\varphi(D)$ is finite, and that φ is safe if it is safe on every D. The safety problem is to determine whether a query is safe, and it is known to be undecidable even for the pure relational calculus [1]. The *state-safety* problem is to decide, for a given φ and D, if φ is safe on D.

We say that safe queries in $\mathrm{RC}(\mathcal{M})$ *have effective syntax* if there exists a recursively enumerable set $\psi_i, i < \omega$, of safe queries in $\mathrm{RC}(\mathcal{M})$ such that, for every SC, every safe $\mathrm{RC}(\mathrm{SC}, \mathcal{M})$ query is equivalent to one of ψ_is. Effective syntax is a first step towards an algebraic language expressing all safe queries. Indeed if such a language exists, safe queries must have effective syntax. That effective syntax exists for safe queries in the pure relational calculus is a classical relational theory result. Other – both positive or negative – results have been proved recently [7, 28].

An important restriction of queries is that to quantification over the active domain. We use quantifiers $\exists x \in \mathrm{adom}$ and $\forall x \in \mathrm{adom}$, whose meaning is as follows: $D \models \exists x \in \mathrm{adom} \varphi(x, \cdot)$ if $D \models \varphi(a, \cdot)$ for some $a \in$ *adom*(D) (as opposed to for some $a \in U$ in the case of the usual $\exists x$ quantifier), and similarly for the universal quantifier. These restricted quantifiers are definable in relational calculus, but it is often helpful to have them available separately.

A relational calculus formula is called an *active-domain* formula if all quantifiers in it are of the form $\forall x \in \mathrm{adom}, \exists x \in \mathrm{adom}$. We say that $\mathrm{RC}(\mathcal{M})$ admits *natural-active collapse* [6] if every $\mathrm{RC}(\mathcal{M})$ formula is equivalent to an active-domain formula. We say that $\mathrm{RC}(\mathcal{M})$ admits *restricted quantifier collapse* if every $\mathrm{RC}(\mathcal{M})$ formula is equivalent to one in which SC-relations appear only under the scope of quantifiers $\exists x \in \mathrm{adom}$ and $\forall x \in \mathrm{adom}$. Note that if $\mathcal{M}$ admits quantifier-elimination, these two notions coincide.

Complexity classes Some complexity results in this paper refer to parallel complexity classes AC^0, TC^0, and NC^1. AC^0 is constant parallel time; more precisely, the class of languages accepted by polynomial-size constant-depth unbounded fan-in circuits. TC^0 additionally has majority gates of unbounded fan-in. In NC^1, there are no majority gates, the depth is allowed to be logarithmic, but fan-in is bounded. It is known that $\mathrm{AC}^0 \subset \mathrm{TC}^0 \subseteq \mathrm{NC}^1$ (parity separates TC^0 from AC^0). We consider uniform versions of these classes [4]; uniform AC^0 over finite structures can be characterized via definability in $\mathrm{FO}(\mathrm{BIT}, <)$: first-order logic with linear order and the BIT predicate ($\mathrm{BIT}(i, j)$ is true iff the jth bit in the binary representation of i is one.) To capture unifom TC^0 it suffices to add counting quantifiers to $\mathrm{FO}(\mathrm{BIT}, <)$ [4].

PH is the polynomial hierarchy, which contains, e.g., NP and coNP and is itself included in PSPACE [24].

As usual, for *data complexity*, one fixes a query Q and considers the complexity of $\{enc(D)\#enc(t) \mid t \in Q(D)\}$. Normally in pure relational calculus the encoding is such that the active domain is considered to be $\{1, \ldots, k\}$, and each number i is represented in binary. When we deal with *interpreted* elements stored in a database, such an encoding is not appropriate, as one needs to take into account operations on those interpreted elements. In particular, in the case of strings over a finite alphabet, we consider the encoding of a string to be itself (in the case of an alphabet different from $\{0, 1\}$ we may have to code letters in $\{0, 1\}$ first).

3 Problematic concatenation

As we said before, most earlier papers considered relational calculus with concatenation RC_{concat}, that is, $\mathrm{RC}(\mathrm{SC}, \langle \Sigma^*, \Omega \rangle)$ where Ω has one operation of concatenation, and constant symbols for each $a \in \Sigma$. This language is extremely attractive in terms of compositionality: given queries Q and Q' returning sets of strings, one can substitute Q and Q' within regular-expressions to form new LIKE queries. However, as noticed in [20], for $\Sigma = \{0, 1, \sharp\}$, RC_{concat} expresses all computable queries on databases containing strings from $\{0, 1\}^*$ (see [27] for a proof). In fact, it is easy to show a more general result:

Proposition 1 *Let Σ contain at least two letters. Then RC_{concat} expresses all computable queries on databases over Σ^*.* $\Box$

In databases, we are accustomed to relational calculus having limited expressiveness; then the queries can be analyzed and often good optimizations can be discovered. This is certainly not the case here; moreover, there is no hope of finding a syntax for safe queries.

Corollary 1 *Let Σ contain at least two letters. Then there is no effective syntax for safe queries in RC_{concat}. Furthermore, the state-safety problem is undecidable for RC_{concat}.* $\Box$

Note that when Σ has one symbol, $\langle \Sigma^*, \cdot \rangle$ is essentially $\langle \mathbb{N}, + \rangle$, and there exists effective syntax for safe queries, and state-safety is decidable [28].

4 Two models of basic string operations

Since RC_{concat} is computationally complete, it is too powerful for use as a query language. We thus examine languages with strictly bounded expressivity and complexity. As mentioned in the introduction, we want to be able to freely compose the output of a string query with a new query, just as we can compose queries in standard relational languages. Hence when we consider a set of operations Ω, we will always close under all first order operations on the structure $\langle \Sigma^*, \Omega \rangle$.

We start by looking at existing SQL string operations. The most often-used operation is `LIKE` pattern-matching. It allows one to say, for example, that a given string is a prefix of another string (by using the `%` pattern meaning "zero or more characters"), and also that a string has a fixed string as a substring. `LIKE` patterns are built from alphabet letters, and characters `%` and `_` (which matches a single letter).

Matching with `LIKE` can be expressed in first-order logic over the the operations $\prec$ and $L_a, a \in \Sigma$: SQL's `LIKE` patterns recognize only star-free languages, and (as we show later) definable subsets of Σ^* in $\langle \Sigma^*, \prec, (L_a)_{a\in\Sigma} \rangle$ are precisely star-free languages. For example, the condition x `LIKE a_b%a_` — saying that the first symbol of x is a, the third is b, and one but last is a again — can be expressed by a formula $\varphi(x)$:

$$\exists u, v, w \left(\begin{array}{ll} & u \prec v \prec w \prec x \\ \wedge & L_a(u) \wedge L_b(v) \wedge L_a(w) \\ \wedge & \psi_1(u) \wedge \psi_3(v) \wedge \psi_{-1}(w) \end{array} \right)$$

where $\psi_1(u), \psi_3(v), \psi_{-1}(w)$ say that u, v, w are prefixes extending up to the first, third, and one but last positions in the string x. Another important operation first-order expressible over $\prec$ and L_a is the *lexicographic* ordering $\leq_{\mathrm{lex}}$. Assume that $\Sigma = \{a_1, \ldots, a_n\}$ and an ordering $a_1 < \ldots < a_n$ is given. The lexicographic ordering $x \leq_{\mathrm{lex}} y$ is then expressed by:

$$x \preceq y \vee \exists z \left(z \prec x \wedge z \prec y \wedge \bigvee_{i<j} ((l_{a_i}(z) \preceq x) \wedge (l_{a_j}(z) \preceq y))\right)$$

Thus, with the basic set of operations $\prec$ and L_a (or their functional version l_a) we can express the two most useful SQL string operations. We therefore begin by considering the first-order query language over $\mathbf{S} = \langle \Sigma^*, \prec, (L_a)_{a\in\Sigma} \rangle$. Note that $\mathbf{S}$ could be equivalently defined as $\langle \Sigma^*, \prec, (l_a)_{a\in\Sigma} \rangle$, as the graph of l_a is definable: $x = l_a(y)$ iff $y \lessdot x \wedge L_a(x)$.

Another SQL string operation is *length* `LEN`. Since this does not return a string, we turn it into a pure string operation that compares lengths of strings: $\mathrm{el}(x, y)$ is true if $|x| = |y|$. Adding this operation to $\mathbf{S}$ results in $\mathbf{S}_{\mathrm{len}} = \langle \Sigma^*, \prec, (L_a)_{a\in\Sigma}, \mathrm{el} \rangle$.

$\mathbf{S}_{\mathrm{len}}$ is a structure well known in model theory and language theory [10, 13]. Consider a few examples of expressibility over $\mathbf{S}_{\mathrm{len}}$. Clearly, other comparisons of string length can be expressed, e.g. $|x| < |y|$ by $\exists z(z \prec y \wedge \mathrm{el}(z, x))$. As a more interesting operation, consider adding symbols on the left of a string: that is, the operation $f_a(x) = a \cdot x$. The graph of this function $F_a = \{(x, y) \mid y = f_a(x)\}$, is definable by

$$\begin{array}{rl} & |y| = |x| + 1 \wedge (\exists w \prec y \ |w| = 1 \wedge L_a(w)) \\ \wedge & \forall z \prec x \exists v \prec y \ (|v| = |z| + 1 \wedge \bigwedge_{b \in \Sigma} L_b(z) \leftrightarrow L_b(v)) \end{array}$$

where $|v| = |u| + 1$ is defined by $\exists w(w \lessdot u \wedge \mathrm{el}(w, v))$.

To summarize, we are dealing with two structures:

- $\mathbf{S} = \langle \Sigma^*, \prec, (l_a)_{a \in \Sigma} \rangle$;
- $\mathbf{S}_{\mathrm{len}} = \langle \Sigma^*, \prec, (l_a)_{a \in \Sigma}, \mathrm{el} \rangle$.

In terms of expressing SQL string operations, $\mathbf{S}$ covers `LIKE`, ordering, as well as substrings of constant length, and `TRIM TRAILING`, that removes all trailing occurrences of a given symbol. $\mathbf{S}_{\mathrm{len}}$ is much more powerful, and covers the `SIMILAR` pattern matching of the SQL3 standard [21] (which is essentially `grep`).

The properties listed below can be found in [13, 10, 8] for $\mathbf{S}_{\mathrm{len}}$ and [8, 10] for $\mathbf{S}$. Note that they are properties of the underlying structures alone, without reference to a database.

Properties of $\mathbf{S}$. Every formula is equivalent to a formula in which quantification is restricted to prefixes of free variables. Moreover, $\mathbf{S}$ has quantifier elimination in the signature extended with the following: a constant symbol, ϵ, for the empty string, the function $x \sqcap y$, and a predicate $P_L(x, y)$ for each star-free language, whose meaning is $x \prec y, y - x \in L$. That is, this signature is infinite, but it only contains unary and binary predicates and functions. The definable subsets of Σ^* are precisely the star-free languages. Neither the function f_a nor the predicate el can be defined over $\mathbf{S}$.

Properties of $\mathbf{S}_{\mathrm{len}}$. It suffices to restrict quantification to strings whose length does not exceed the length of free variables. $\mathbf{S}_{\mathrm{len}}$ does not have quantifier elimination in any reasonable relational signature (that is, in any signature that has an upper bound on the arity of predicates). The class of subsets of Σ^* definable over $\mathbf{S}_{\mathrm{len}}$ is precisely the class of regular languages (thus, grep pattern-matching is definable in $\mathbf{S}_{\mathrm{len}}$).

5 Expressive power and complexity

In this section we study expressiveness and complexity over $\mathbf{S}$ and $\mathbf{S}_{\mathrm{len}}$.

5.1 Relational calculus over S

Our goal here is to get bounds on the expressiveness and data complexity for queries in $\mathrm{RC}(\mathbf{S})$. The main tool used is a collapse result, Theorem 1, in the spirit of those produced for constraint databases [6, 5]. Recall that relational calculus over a domain $\mathrm{RC}(\mathcal{M})$ admits *restricted quantifier collapse* if every $\mathrm{RC}(\mathrm{SC}, \mathcal{M})$ formula $\varphi(\vec{x})$ is equivalent to a formula $\varphi'(\vec{x})$ in which SC-predicates occur only within the scope of active domain quantifiers $\exists x \in \mathrm{adom}$ and $\forall x \in \mathrm{adom}$.

To prove this, we first prove a simple proposition saying that it suffices to quantify over prefixes of the active domain. Extend $\mathrm{RC}(\mathrm{SC}, \mathbf{S})$ with quantifiers of the form $\exists x \preceq \mathrm{adom}$ and $\forall x \preceq \mathrm{adom}$, whose meaning is as follows. Given a formula $\varphi(x, \vec{y})$, an interpretation $\vec{a}$ for $\vec{y}$, and a database D, $\exists x \preceq \mathrm{adom}\ \varphi(x, \vec{a})$ states that there exists a string c making $\varphi(c, \vec{a})$ true such that either $c \preceq a_i$ for a_i a component of $\vec{a}$, or $c \preceq b$ where b is in $adom(D)$. When there is no database, only the first of the previous cases is relevant, and in this case we are just saying that bounded quantification suffices.

Proposition 2 *Every* $\mathrm{RC}(\mathrm{SC}, \mathbf{S})$ *formula is equivalent to a formula that only uses quantifiers* $\exists x \preceq \mathrm{adom}$ *and* $\forall x \preceq \mathrm{adom}$.

Proof sketch. We write $(D_1, \vec{s}_1) \equiv_k (D_2, \vec{s}_2)$ if the duplicator has a winning strategy in the k-round Ehrenfeucht-Fraïssé game on $\mathbf{S}$ augmented with SC-relations and constants interpreted as $(D_1, \vec{s}_1)$ and $(D_2, \vec{s}_2)$. We write $(D_1, \vec{s}_1) \equiv_k^b (D_2, \vec{s}_2)$ if the duplicator has a winning strategy in the same game restricted to $\mathit{prefix}(D_1) \cup \mathit{prefix}(\vec{s}_1)$ and $\mathit{prefix}(D_2) \cup \mathit{prefix}(\vec{s}_2)$. We then show that $\equiv_{k+m+1}^b$ refines $\equiv_k$, where m is the maximum arity of a relation in SC. This is because the winning strategy for the duplicator over $\mathbf{S}$ and SC is determined by the winning strategy on the restriction to prefixes of the strings in SC-relations and free variables. This implies the result. □

Using this, and techniques similar to those in [6], we can show (in a constructive way) the following result:

Theorem 1 $\mathrm{RC}(\mathbf{S})$ *admits restricted quantifier collapse.* □

By showing that quantification can be bounded by relations, Theorem 1 gives the intuition that an $\mathrm{RC}(\mathbf{S})$ query can be transformed into an ordinary SQL query over `LIKE`: this will be made precise in Section 6. Here we note that a a straightforward corollary of Theorem 1 shows that the data complexity for $\mathrm{RC}(\mathbf{S})$ matches that of pure relational calculus.

Corollary 2 *The data complexity of* $\mathrm{RC}(\mathbf{S})$ *is in* AC^0. *In particular, neither parity nor connectivity test is expressible in* $\mathrm{RC}(\mathbf{S})$.

Another corollary concerns the expressive power of generic queries. Recall that a query is *generic* if it commutes with permutations on the domain; in other words, it is independent of specific elements stored in a database. Combining Theorem 1 with the active generic collapse [6], we obtain:

Corollary 3 *Every generic query expressible in* $\mathrm{RC}(\mathbf{S})$ *is already expressible in* $\mathrm{RC}(<)$*, relational calculus over ordered databases.* □

With respect to time complexity Corollary 2 only gives a polynomial upper bound. We show next that for unary databases we get a much stricter complexity result. We call a database schema SC *unary* if it only contains unary relation names.

Proposition 3 *For unary* SC*, Boolean* $\mathrm{RC}(\mathrm{SC}, \mathbf{S})$*-queries can be evaluated in linear time in the size of the database.* □

5.2 Relational calculus over $\mathbf{S}_{\mathrm{len}}$

We have seen nice that query evaluation for relational calculus over $\mathbf{S}$ has low complexity. However, many useful queries of low complexity, such as the query that appends a fixed string on the left of a given column, are not expressible in $\mathbf{S}$. Hence we examine the addition of the equal length predicate, that is, relational calculus over $\mathbf{S}_{\mathrm{len}}$. Throughout this section, we assume that the alphabet has at least two symbols (as over the one-symbol alphabet, equal length is simply equality and thus does not give us any extra power).

To analyze the expressive power and complexity of $\mathbf{S}_{\mathrm{len}}$, we again make use of a normal-form result for queries. In this case it is no longer sufficient to quantify over prefixes of strings in the active domain; however a different restricted quantification suffices.

We introduce quantifiers $\exists\,|x| \le \mathrm{adom}$ and $\forall\,|x| \le \mathrm{adom}$ to be interpreted as follows. Given a formula $\varphi(\vec{y})$, a database D and an interpretation $\vec{a}$ for $\vec{y}$, a subformula $\exists\,|x| \le \mathrm{adom}\ \alpha(x,\cdot)$ is satisfied if there exists a string c satisfying $\alpha(c,\cdot)$ such that the length of c does not exceed the length of the longest string in $adom(D)$ and $\vec{a}$. We call these *length-restricted quantifiers.* Note that they are just a notational convenience, as they can be expressed in $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$. Moreover, they capture the expressiveness of $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$:

Proposition 4 *Every* $\mathrm{RC}(\mathrm{SC}, \mathbf{S}_{\mathrm{len}})$ *formula is equivalent to a formula that uses only length-restricted quantifiers.*

Proof sketch. The proof is along the lines of the proof of Proposition 2, but for a finite structure one takes the restriction based on length rather than prefixes. □

Prefix-restricted quantification does not suffice for $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$. Indeed, consider the following query Q on a unary relation U: $Q(U)$ is true iff U contains a single element, which is from 0^* and of even length. This is expressible in $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$ by

$$\exists! x\ U(x) \wedge \forall x (U(x) \to (x \in 0^*) \wedge \exists z \in (01)^* \mathrm{el}(z, x)).$$

Note that the predicates $x \in 0^*$ and $z \in (01)^*$ can be expressed even over $\mathbf{S}$: recall that $\mathbf{S}$ can define any star-free language and $\mathbf{S}_{\mathrm{len}}$ any regular language. However, this query Q is inexpressible with just prefix quantification: if it were, then over single-element databases contained in 0^*, el could be eliminated from the query. Hence the set of strings from 0^* of even length would be definable over $\mathbf{S}$. But this language is not star-free, and this contradicts the fact that the languages definable over $\mathbf{S}$ are exactly the star-free languages [8].

As with Theorem 1, Proposition 4 gives us an upper bound on the complexity of $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$:

Corollary 4 *The data complexity of* $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$ *is in PH.*

Proof sketch. To check if $D \models \varphi(\vec{a})$, it is enough to quantify over strings whose length does not exceed N, where N is the maximum length of a string in $adom(D) \cup \vec{a}$ (see Proposition 4). If φ has alternation depth k this can be done by a polynomial time alternating Turing machine with k alternations, hence in PH. □

One can also derive an upper bound on generic computation, albeit not as low as for $\mathbf{S}$. A *relational (Boolean) query* is a set of isomorphism types of SC-databases (w.r.t. the SC-relations only). A relational query is in AC^0 if it is in AC^0 under the usual relational encoding enc_0: elements of a k-element active domain are encoded by $1, \ldots, k$, in binary (cf. [1]). A relational query Q is expressible in $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$ if there is a $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$ sentence Φ such that the SC-isomorphism type of D is in Q iff $D \models \Phi$.

Theorem 2 *Any relational query that is expressible in* $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$ *is in* AC^0. *Thus, parity test and connectivity test are not definable in* $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$. □

We now prove lower bounds that show the complexity of $\mathbf{S}_{\text{len}}$ queries, although within PH, may be prohibitively high. Let MSO(SC) be the class of queries over SC expressible in monadic second-order logic. This includes queries of high-complexity, namely for each level of the polynomial hierarchy, PH, complete queries [2], in particular, NP-complete and coNP-complete ones (3-colorability and its complement). Such queries cannot be expressed over arbitrary databases in $\mathrm{RC}(\mathbf{S}_{\text{len}})$; however, they can be expressed under some additional assumptions.

We say that the *width* of the active domain of a SC database D (over Σ^*) is k if k is the maximal size of a subset of $adom(D)$ whose elements are pairwise comparable by the prefix relation. It should be noted that every database D can be transformed into a database D' of width 1 which is isomorphic to D with respect to the SC-predicates.

Proposition 5 *For every fixed k, all* MSO(SC)*-expressible queries can be expressed over databases of width at most k in* $\mathrm{RC}(\mathrm{SC}, \mathbf{S}_{\text{len}})$. □

Thus, while not computationally complete as RC_{concat}, $\mathrm{RC}(\mathbf{S}_{\text{len}})$ can express some queries that one would not normally expect to be expressible in a first-order language.

Recall that we had a linear time bound for the evaluation of Boolean $\mathrm{RC}(\mathbf{S})$-queries on unary databases. This might not be the case for $\mathrm{RC}(\mathbf{S}_{\text{len}})$. Even worse, there might be even no fixed polynomial bound. Indeed it is possible to show that any graph query in $\mathrm{RC}(\mathbf{S}_{\text{len}})$ can be encoded by a unary query, where the input to the unary query is computed in polynomial time from the input graph.

Thus, a linear (or fixed polynomial) bound for the evaluation of Boolean $\mathrm{RC}(\mathbf{S}_{\text{len}})$-queries on unary databases would imply a fixed polynomial bound for the data complexity of first-order sentences on ordered graphs. It would imply further a fixed polynomial bound for the evaluation of first-order sentences on BIT-structures (cf., [3]). This, in turn, would separate first-order logic from least fixed point logic on such structures and therefore imply the validity of the *ordered conjecture* [25] with various consequences in complexity theory (see [3] for a discussion).

6 Safe Queries

Both $\mathrm{RC}(\mathbf{S})$ and $\mathrm{RC}(\mathbf{S}_{\text{len}})$ contain queries that sometimes produce infinite output. Thus one of our goals is to syntactically capture the safe queries in these languages, and to be able to analyze safety properties of a query – for example, given an arbitrary query and a database, to tell whether the output of the query on that database is finite. We saw that this cannot be done if the set of operations includes concatenation. In contrast, we will show that for $\mathrm{RC}(\mathbf{S})$ and $\mathrm{RC}(\mathbf{S}_{\text{len}})$ we can syntactically describe safe queries, give an algebra that captures these queries, and extend the major decidability results for query safety analysis that hold for pure relational calculus.

6.1 Effective syntax for safe queries

The simplest way to show that queries in $\mathrm{RC}(\mathcal{M})$ have effective syntax is to show that one can test if a given query returns a finite result on a given database. To do so, it is enough to ensure that *finiteness is definable* in $\mathrm{RC}(\mathcal{M})$. Formally, finiteness is definable in $\mathrm{RC}(\mathcal{M})$ if there exists a sentence Φ^{safe} in the language of $\mathcal{M}$ and SC expanded with a single new unary predicate symbol U such that for any query $\varphi(x)$ and any database D, $(D, \varphi(D)) \models \Phi^{\text{safe}}$ iff $\varphi(D)$ is finite. For example, finiteness is easily definable in $\mathrm{RC}(\mathbf{S}_{\text{len}})$ by

$$\exists y \forall x (U(x) \rightarrow \exists z \prec y \ \mathrm{el}(z, x)).$$

Once finiteness is definable, an enumeration of safe queries can easily be obtained. Given a query $\varphi(\vec{x})$, let $\psi_\varphi(x)$ be another relational calculus query that defines the active domain of the output of φ. Let $\Phi^{\text{safe}}_\varphi$ be the Boolean query obtained from Φ^{safe} be replacing $U(\cdot)$ by $\psi_\varphi(\cdot)$. Then $\varphi(\vec{x}) \wedge \Phi^{\text{safe}}_\varphi$ lists all safe queries.

For traditional relational calculus, and for its analogs over order constraints, linear constraints, and polynomial constraints, finiteness can easily be shown to be definable [7]. It is thus surprising that for $\mathrm{RC}(\mathbf{S})$ this approach does not work:

Proposition 6 *Finiteness is not definable in* $\mathrm{RC}(\mathbf{S})$.

Proof sketch. This is proved using an Ehrenfeucht-Fraïssé game argument. It shows that, for every k, there exist K, m such that a database containing all strings of length at most K cannot be distinguished with only k moves from a database containing the infinite set of strings $(0^m 1^m)^*$ together with all the strings of the form $(0^m 1^m)^* w$, where w has length at most $K + 2m$. □

While post-checking finiteness is a way to obtain effective syntax for safe queries, one often wishes to have a more explicit representation of safe queries. It turns out that we can get natural representations for safe queries in RC($\mathbf{S}$) and RC($\mathbf{S}_{\mathrm{len}}$). The technique we use derives from work on safe languages with linear or polynomial constraints [7]: for each query Q, we effectively construct another *safe* query Q' that gives an upper bound on $Q(D)$, if it is finite. Such explicit constructions are used to prove the theorem below, as well as to provide relational algebra extensions.

We follow the idea of range-restriction as presented in [7]. A formula $\gamma(x, z)$ over $\mathcal{M}$ is called *algebraic* if for every b, the set $\{a \mid \mathcal{M} \models \gamma(a, b)\}$ is finite. An RC($\mathcal{M}$) query in *range-restricted form* is a pair $Q = (\gamma(x, y), \varphi(x_1, \ldots, x_n))$, where φ is an arbitrary query and γ is an algebraic formula over $\mathcal{M}$. The semantics is given by $\varphi(\vec{x}) \wedge \exists \vec{y} \in \text{adom } (\bigwedge_i \gamma(x_i, y_i))$. That is,

$$Q(D) \quad = \quad \gamma(adom(D))^n \cap \varphi(D)$$

where $\gamma(X) = \{a \mid \gamma(a, b) \text{ for some } b \in X\}$. Clearly, every query in range-restricted form is safe.

Theorem 3 *Let $\mathcal{M}$ be $\mathbf{S}$ or $\mathbf{S}_{\mathrm{len}}$. Then there is a recursive set Γ of algebraic formulae over $\mathcal{M}$ such that, given a query $\varphi(\vec{x})$ in RC($\mathcal{M}$), there is $\gamma(x, y) \in \Gamma$ with the property that the range-restricted query $Q = (\gamma, \varphi)$ coincides with φ on all databases over which φ is safe.*

Proof sketch. The proof is based on two lemmas, which show that if a query $\varphi(x)$ is satisfied by an element that is sufficiently far from $adom(D)$, then φ returns an infinite result on D. The proof of these lemmas relies on a kind of pumping Lemma which permits to derive infinitely many strings as soon as a big enough one is found. Define $d(s, C)$ as $|s| - |s \sqcap C|$, and $\downarrow D = \{s \mid |s| \leq |s'|, s' \in adom(D)\}$.

Lemma 1 *Let $\varphi(x)$ be a RC($\mathbf{S}$) query. Then there exists (and can be effectively found if φ only uses prefix-restricted quantification) a number $k > 0$, such that the following holds. Assume that $D \models \varphi(s)$ for some s with $d(s, prefix(D)) > k$. Then there are infinitely many strings c such that $D \models \varphi(c)$.* □

Lemma 2 *Let $\varphi(x)$ be a RC($\mathbf{S}_{\mathrm{len}}$) query. Then there exists (and can be effectively found if φ only uses length restricted quantification), a number $k > 0$ such that the following holds. Assume that $D \models \varphi(s)$ for some s with $d(s, \downarrow D) > k$. Then there are infinitely many strings c such that $D \models \varphi(c)$.*

To prove the theorem, take an arbitrary query $\psi(\vec{y})$ and form $\varphi(x)$ that defines the active domain of the output of ψ. It then suffices to prove the theorem for $\varphi(x)$, since ψ is safe for D iff φ is safe for D, and thus for any γ such that (γ, φ) is equivalent to φ on all D for which φ is safe, the same would be true for (γ, ψ) and ψ.

Having reduced the problem to queries in one variable, simply apply the corresponding lemmas. For RC($\mathbf{S}$), given $\varphi(x)$, find the number k as in Lemma 1, and let $\gamma(x, y)$ say that x is a prefix of the string of the form $y \cdot s$ with $|s| \leq k$. From Lemma 1 it follows that (γ, φ) is equivalent to φ on any D for which φ is safe. Finally, γ is clearly algebraic, and expressible over $\mathbf{S}$ for any fixed k.

For RC($\mathbf{S}_{\mathrm{len}}$), given $\varphi(x)$, we get k from Lemma 2 and let $\gamma(x, y)$ be a $\mathbf{S}_{\mathrm{len}}$ formula saying that the length of x is at most the length of y plus k. Clearly, this is expressible for each fixed k, and (γ, φ) coincides with φ on any D for which φ is safe. This completes the proof of the theorem. □

Corollary 5 *For both* RC($\mathbf{S}$) *and* RC($\mathbf{S}_{\mathrm{len}}$)*, the classes of range-restricted and safe queries coincide, and safe queries have effective syntax.* □

Note that for queries in RC($\mathbf{S}$) and RC($\mathbf{S}_{\mathrm{len}}$) that use a restricted form of quantification (prefix or length), the proof gives us a stronger result: namely, the formula γ can be effectively found for a given φ.

6.2 Relational algebras

It is a classical result of relational database theory that the set of safe relational calculus queries is precisely the set of relational algebra queries. This result extends to string calculi considered here: safety theorems proved earlier can be used to show that safe queries in RC($\mathbf{S}$) and RC($\mathbf{S}_{\mathrm{len}}$) can be captured by appropriate extensions of relational algebra.

Let safe_RC($\mathcal{M}$) be the class of all safe queries in RC($\mathcal{M}$). To define algebras capturing safe_RC($\mathcal{M}$) for the previous two structures, we need a number of operations extending the usual relational algebra (that is, $\sigma, \pi, \times, -, \cup$):

R_ϵ: is a constant unary relation $\{\epsilon\}$.

σ_α: for a formula $\alpha(x_1, \ldots, x_n)$. On an n-attribute relation R, it returns the set of tuples $(s_1, \ldots, s_n)$ from R such that $\alpha(s_1, \ldots, s_n)$ holds.

$\mathtt{prefix}_i$: On an m-attribute relation R, it returns $(m+1)$-attribute relation $\{(s_1, \ldots, s_{m+1}) \mid (s_1, \ldots, s_m) \in R, s_{m+1} \preceq s_i\}$.

$\mathtt{addl}_i^a$, $a \in \Sigma$: On an m-attribute relation R, it returns the $(m+1)$-attribute relation $\{(s_1, \ldots, s_{m+1}) \mid (s_1, \ldots, s_m) \in R, s_{m+1} = s_i \cdot a\}$.

$\downarrow_i$: Given an m-attribute relation R, $\downarrow_i (R)$ returns $\{(s_1,\ldots,s_{m+1}) \mid (s_1,\ldots,s_m) \in R, |s_{m+1}| \leq |s_i|\}$.

It should be pointed out that the formula α in σ_α does not refer to the database.

We now define the relational algebras:

RA(**S**) extends relational algebra with R_ϵ, σ_α, where α ranges over FO(**S**) formulae, $\texttt{prefix}_i$ and $\texttt{addl}_i^a$.

RA($\mathbf{S}_{\text{len}}$) extends relational algebra with R_ϵ, σ_α, where α ranges over FO($\mathbf{S}_{\text{len}}$) formulae, $\downarrow_i$, $\texttt{prefix}_i$, and $\texttt{addl}_i^a$.

Theorem 4
- safe_RC(**S**) = RA(**S**);
- safe_RC($\mathbf{S}_{\text{len}}$) = RA($\mathbf{S}_{\text{len}}$).

Proof sketch. Theorems 3 showed that there is a bound on outputs of safe queries. To prove the theorem, it suffices to notice that those bounds can be computed by relational algebra expressions. □

One of the operations in RA($\mathbf{S}_{\text{len}}$), $\downarrow_i$, is very expensive, as it may create sets whose size is exponential in the size of the input. It is, however, unavoidable, as there are very expensive (e.g., NP-complete) safe queries in RC($\mathbf{S}_{\text{len}}$).

6.3 Deciding Safety Properties of Queries

Although query safety is undecidable for pure relational calculus (and hence for any extension), state-safety (given a query φ and a database D, is $\varphi(D)$ finite?) is decidable. State safety is also known to be decidable for various extensions of the form RC($\mathcal{M}$) (for example, for the natural numbers with successor [28] or the real field [7]). For RC(**S**) and RC($\mathbf{S}_{\text{len}}$), this decidability holds as well:

Proposition 7 *State-safety is decidable for* RC(**S**) *and* RC($\mathbf{S}_{\text{len}}$). □

As query safety is undecidable, one often considers restrictions for which decidability can be obtained. Here we look at one of the most fundamental classes of queries – *conjunctive queries*. We take their definition in the context of interpreted operations from [7, 23]. A conjunctive query in RC($\mathcal{M}$) is a query of the form

$$\varphi(\vec{x}) \;\equiv\; \exists \vec{y} \bigwedge_{i=1}^{k} S_i(\vec{u}_i) \;\wedge\; \gamma(\vec{x},\vec{y}),$$

where $k \geq 0$, each S_i is a schema relation, $\vec{u}_i$ is a subtuple of $(\vec{x},\vec{y})$ of the same arity as S_i, and γ is an $\mathcal{M}$ formula. A Datalog-like notation for such a query would be $\varphi(\vec{x})$:- $S_1(\vec{u}_1),\ldots,S_k(\vec{u}_k),\gamma(\vec{x},\vec{y})$.

In [7], safety of conjunctive queries was shown decidable for RC($\mathcal{M}$), for various structures $\mathcal{M}$ on the reals with numerical operations. We now show a general result from which the decidability results for string structures $\mathbf{S}, \mathbf{S}_{\text{len}}$ and those considered in [7] follow. We say that *finiteness is definable with parameters* in $\mathcal{M}$ if for each formula $\psi(\vec{x},\vec{y})$ in $\mathcal{M}$, there exists and can be effectively found another formula $\psi_{\text{fin}}(\vec{y})$ such that $\mathcal{M} \models \psi_{\text{fin}}(\vec{a})$ iff the set $\{\vec{b} \mid \mathcal{M} \models \psi(\vec{b},\vec{a})\}$ is finite.

Theorem 5 *Assume that $\mathcal{M}$ can be expanded to $\mathcal{M}'$ such that the theory of $\mathcal{M}'$ is decidable, and finiteness is definable with parameters in $\mathcal{M}'$. Then safety of Boolean combinations of conjunctive queries in* RC($\mathcal{M}$) *is decidable.* □

We know that Th($\mathbf{S}_{\text{len}}$) is decidable [10]. Moreover, finiteness is definable with parameters: for $\psi(\vec{x},\vec{y})$, $\psi_{\text{fin}}(\vec{y})$ is $\exists \vec{u}(\forall \vec{x}\psi(\vec{x},\vec{y}) \;\rightarrow\; \exists \vec{z} \bigwedge_i z_i \;\prec\; u_i \;\text{el}(z_i,x_i))$. Thus:

Corollary 6 *The safety of Boolean combinations of conjunctive queries in* RC(**S**) *and* RC($\mathbf{S}_{\text{len}}$) *is decidable.* □

7 Tame extensions of RC(S)

In the previous two sections we considered two different relational calculi for databases with strings: RC(**S**) and RC($\mathbf{S}_{\text{len}}$). The former models operations such as the `LIKE` pattern-matching and lexicographic ordering; the latter adds length comparisons, and enables additional operations such as trimming/adding symbols on both left and right of a string, and the `SIMILAR` pattern-matching for checking membership in a regular language. Both languages have some nice properties: for example, there is effective syntax, and even an algebra, for safe queries. However, RC(**S**) misses a number of important string functions, while the complexity of RC($\mathbf{S}_{\text{len}}$) can be quite high: we saw how to encode NP and coNP-complete problems on inputs of a special kind.

Thus, a natural question is whether one can add operations to RC(**S**) while maintaining its nice properties: effective syntax for safe queries and low data complexity. We give here a positive answer to this question, by considering two extensions. The first one gives us operations for adding/trimming symbols on the left; for example, $\texttt{TRIM}_a(s)$, where $a \in \Sigma$, produces s' if $s = a \cdot s'$,

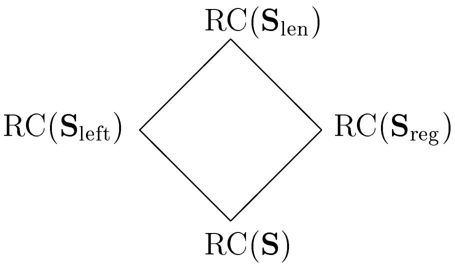

Figure 1: Relationships between $\mathrm{RC}(\mathbf{S}), \mathrm{RC}(\mathbf{S}_{\mathrm{left}}), \mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$, and $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$.

and ϵ if the first symbol of s is not a. The other extension is by allowing tests for membership in a regular language, without the full power of the equal length predicate. We show that both extensions share most of its properties with $\mathrm{RC}(\mathbf{S})$, while adding significantly to the expressiveness of the language.

The first operation we consider is adding one single character on the left: $s \mapsto a \cdot s$, and its inverse $\mathtt{TRIM}_a(s)$ denoted by $s - a$. That is, we consider the structure:

- $\mathbf{S}_{\mathrm{left}} = \langle \Sigma^*, \prec, (l_a)_{a\in\Sigma}, (f_a)_{a\in\Sigma}\rangle$.

This is a proper extension of $\mathbf{S}$, as the graph of the function f_a, $\{(s, f_a(s)) \mid s \in \Sigma^*\}$, is not definable over $\mathbf{S}$ [8]. The graph of the subtraction operation is definable with f_a. We also remark that while the classes of subsets of $(\Sigma^*)^k$, $k > 1$, definable in $\mathbf{S}$ and $\mathbf{S}_{\mathrm{left}}$ are different, over both structures the class of definable subsets of Σ^* is the same, that is, the class of star-free languages [8].

The second extension we consider allows us to model more general regular expression pattern-matching. Of course any regular language is definable over $\mathbf{S}_{\mathrm{len}}$, and thus such pattern-matching can be done in the more complex model $\mathrm{RC}(\mathbf{S}_{\mathrm{len}})$.

We will add regular expression pattern-matching directly to $\mathbf{S}$, without adding the equal length predicate. Recall that $\mathbf{S}$ has quantifier elimination in the extension that includes predicates $P_L(x, y)$, for each star-free language, whose meaning is $x \prec y$ and $y - x \in L$. We now define $\mathbf{S}_{\mathrm{reg}}$ to be the extension of $\mathbf{S}$ with all such predicates when L ranges over *regular* languages. Note that membership of x in any regular language L is definable by $P_L(\epsilon, x)$. To summarize, we are dealing with $\mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$ where

- $\mathbf{S}_{\mathrm{reg}} = \langle \Sigma^*, \prec, (L_a)_{a\in\Sigma}, (P_L)_{L \text{ regular}}\rangle$.

Every set definable in $\mathbf{S}_{\mathrm{reg}}$ is definable in $\mathbf{S}_{\mathrm{len}}$ (as $\mathbf{S}_{\mathrm{len}}$ expresses all predicates P_L), but the converse is not true since the equal length predicate is not definable in $\mathbf{S}_{\mathrm{reg}}$ [8]. Furthermore, the class of subsets of Σ^* definable in $\mathbf{S}_{\mathrm{reg}}$ is exactly the class of regular languages.

$\mathrm{RC}(\mathbf{S}_{\mathrm{left}})$ and $\mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$ are incomparable in term of expressive power. Indeed a simple game argument shows that the relation $\{(x, y) \mid y = f_a(x)\}$ is not definable in $\mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$. Moreover, any language which is not star-free is not definable in $\mathrm{RC}(\mathbf{S}_{\mathrm{left}})$ [8]. Figure 1 summarizes the inclusion relationships between the various relational calculi introduced in this paper, the higher ones being more expressive.

We start with expressive power. Both $\mathrm{RC}(\mathbf{S}_{\mathrm{left}})$ and $\mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$ behave similarly to $\mathrm{RC}(\mathbf{S})$:

Theorem 6 $\mathrm{RC}(\mathbf{S}_{\mathrm{left}})$ *and* $\mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$ *admit the restricted quantifier collapse.*

Proof sketch. Let $\mathbf{S}^+_{\mathrm{left}}$ be the expansion of $\mathbf{S}_{\mathrm{left}}$ with the following (definable) predicates and functions: a constant symbol, ϵ, for the empty string, the binary function $\sqcap$ for the longest common prefix, the predicate $P_L(x, y)$ for each star-free language L, and the function $x \mapsto x - a$, for each $a \in \Sigma$. Let $\mathbf{S}^+_{\mathrm{reg}}$ be the expansion of $\mathbf{S}_{\mathrm{reg}}$ with the (definable) function $\sqcap$ and a constant ϵ for the empty string .

It is shown in [8] that $\mathbf{S}^+_{\mathrm{left}}$ and $\mathbf{S}^+_{\mathrm{reg}}$ have quantifier elimination, and the isolation property which are known to imply the collapse [5, 16]. $\square$

Corollary 7 $\mathrm{RC}(\mathbf{S}_{\mathrm{left}})$ *queries have* AC^0 *data complexity, and* $\mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$ *queries have* NC^1 *data complexity. Furthermore, every generic query expressible in* $\mathrm{RC}(\mathbf{S}_{\mathrm{left}})$ *or* $\mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$ *is expressible in* $\mathrm{RC}(<)$. $\square$

As for query safety, several results extend straightforwardly to $\mathrm{RC}(\mathbf{S}_{\mathrm{left}})$ and $\mathrm{RC}(\mathbf{S}_{\mathrm{reg}})$. Since all operations of $\mathbf{S}_{\mathrm{left}}$ and $\mathbf{S}_{\mathrm{reg}}$ are expressible over $\mathbf{S}_{\mathrm{len}}$, the proof of Proposition 7 and Theorem 5 give us

Model	Data complexity	Data complexity of generic queries	Effective syntax for safe queries	Relational algebra	Safety of CQ
RC($\mathbf{S}$)	AC^0	FO($<$)	yes	yes	decidable
RC($\mathbf{S}_{\rm len}$)	PH	AC^0	yes	yes	decidable
RC($\mathbf{S}_{\rm left}$)	AC^0	FO($<$)	yes	yes	decidable
RC($\mathbf{S}_{\rm reg}$)	NC^1	FO($<$)	yes	yes	decidable
RC$_{concat}$	undecidable	undecidable	no	no	undecidable

Figure 2: Summary of the results

Corollary 8 *The state-safety problem and the safety of Boolean combinations of conjunctive queries are decidable for both* RC($\mathbf{S}_{\rm left}$) *and* RC($\mathbf{S}_{\rm reg}$). $\square$

The effective syntax result can be proved for both RC($\mathbf{S}_{\rm left}$) and RC($\mathbf{S}_{\rm reg}$), but considerably more work is needed (especially in the case of $\mathbf{S}_{\rm left}$).

Theorem 7 *Let $\mathcal{M}$ be $\mathbf{S}_{\rm left}$ or $\mathbf{S}_{\rm reg}$. There exists a recursive collection of algebraic formulae $\Gamma = \{\gamma_i(x,y)\}_{i\in\omega}$ over $\mathcal{M}$ such that for every* RC($\mathcal{M}$) *query $\varphi(\vec{x})$, there is an algebraic formula $\gamma_i(x,y) \in \Gamma$ with the property that the range-restricted query $Q = (\gamma_i, \varphi)$ coincides with φ on all databases over which φ is safe.*

Proof sketch. As for $\mathbf{S}$ and $\mathbf{S}_{\rm len}$, we show how to construct upper bounds on outputs of safe queries. The technical details are rather long and are available in the full version of the paper [9]. $\square$

Corollary 9 *For both* RC($\mathbf{S}_{\rm left}$) *and* RC($\mathbf{S}_{\rm reg}$), *the classes of range-restricted and safe queries coincide, and safe queries have effective syntax.* $\square$

7.1 Relational algebras

We can likewise capture safety for RC($\mathbf{S}_{\rm left}$) and RC($\mathbf{S}_{\rm reg}$) with relational algebras. For that, we need the following operations (in addition to $\pi, \sigma, \times, -, \cup$, and R_ϵ, $\texttt{prefix}$, $\texttt{addl}$, $\downarrow$ used in the definition of RA($\mathbf{S}$) and RA($\mathbf{S}_{\rm len}$)):

$\texttt{addf}_i^a$, $a \in \Sigma$: On an m-attribute relation R, it returns an $m+1$-attribute relation that holds the tuples $\{(s_1,\ldots,s_{m+1}) \mid (s_1,\ldots,s_m) \in R, s_{m+1} = a \cdot s_i\}$.

$\texttt{trim}_i^a$, $a \in \Sigma$: On an m-attribute relation R, it returns an $m+1$-attribute relation that holds the tuples $\{(s_1,\ldots,s_{m+1}) \mid (s_1,\ldots,s_m) \in R, s_{m+1} = s_i - a\}$.

We now define relational algebras:

RA($\mathbf{S}_{\rm left}$) is the extension of relational algebra with σ_α (where α ranges over $\mathbf{S}_{\rm left}$ formulae), $\texttt{prefix}$, $\texttt{addf}_i^a$ and $\texttt{trim}_i^a$.

RA($\mathbf{S}_{\rm reg}$) extends relational algebra with R_ϵ, σ_α, where α ranges over FO($\mathbf{S}_{\rm reg}$) formulae, $\texttt{prefix}_i$ and $\texttt{addl}_i^a$.

Theorem 8
- safe_RC($\mathbf{S}_{\rm left}$) = RA($\mathbf{S}_{\rm left}$);
- safe_RC($\mathbf{S}_{\rm reg}$) = RA($\mathbf{S}_{\rm reg}$).

8 Conclusion

We have studied extensions of the standard relational calculus with various sets of string operations. We were interested in languages that were not computationally complete, but rather shared the attractive complexity-theoretic and static-analysis properties of relational calculus.

The language RC($\mathbf{S}$) can be seen as a nice foundation over which other languages should be built. It covers the most rudimentary string operations, but its expressive power is quite limited. The extension RC($\mathbf{S}_{\rm len}$) that allows string-length comparisons is too powerful (but still not computationally complete). We therefore considered two languages in between, that can express some important operations found in RC($\mathbf{S}_{\rm len}$), but still have low data complexity, effective syntax for safe queries, and corresponding relational algebras. The main results on these relational calculi are summarized in Figure 2.

Regarding further research, it would be interesting to study an extension of RC($\mathbf{S}$) in the spirit of RC($\mathbf{S}_{\rm left}$) by allowing inserting characters at arbitrary position in a string x, specified by a prefix of x.

Acknowledgments We thank Wolfgang Thomas, Scott Weinstein, and Emmanuel Waller for fruitful discussions on the subject.

References

[1] S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] M. Ajtai, R. Fagin and L. Stockmeyer. The closure of monadic NP. In *STOC '98*, pages 309–318.

[3] A. Atserias, Ph. Kolaitis. First-order logic vs. fixed-point logic in finite set theory. In *LICS'98*, pages 275–284.

[4] D.A. Barrington, N. Immerman, H. Straubing. On uniformity within NC^1. *JCSS*, 41:274–306,1990.

[5] O. Belegradek, A. Stolboushkin, M. Taitslin. Extended order-generic queries. *Annals of Pure and Applied Logic* 97 (1999), 85–125.

[6] M. Benedikt, L. Libkin. Relational queries over interpreted structures. *J. ACM* 47 (2000), 644–680.

[7] M. Benedikt, L. Libkin. Safe constraint queries. *SIAM J. Comput.* 29 (2000), 1652–1682.

[8] M. Benedikt, L. Libkin, T. Schwentick, L. Segoufin. A model-theoretic approach to regular string relations. Technical report, INRIA, 2000.

[9] M. Benedikt, L. Libkin, T. Schwentick, L. Segoufin. String operations in query languages. Technical report, INRIA, 2000.

[10] A. Blumensath and E. Grädel. Automatic structures. In *LICS'00*, pages 51–62.

[11] A. Bonner and G. Mecca. Sequences, datalog, and transducers. *JCSS* 57 (1998), 234–259.

[12] A. Bonner and G. Mecca. Querying string databases with transducers. In *DBPL'97*, pages 118–135.

[13] V. Bruyère, G. Hansel, C. Michaux, R. Villemaire. Logic and p-recognizable sets of integers. *Bull. Belg. Math. Soc.* 1 (1994), 191–238.

[14] M. Consens and T. Milo. Algebras for querying text regions: expressive power and optimization. *JCSS* 57 (1998), 272–288.

[15] E. Dantsin, A. Voronkov. Expressive power and data complexity of query languages for trees and lists. In *PODS'2000*, pages 157–165.

[16] J. Flum and M. Ziegler. Pseudo-finite homogeneity and saturation. Preprint, Freiburg University, 1998.

[17] S. Ginsburg and X.S. Wang. Pattern matching by rs-operations: toward a unified approach to querying sequenced data. In *PODS'92*, pages 293–300.

[18] G. Grahne and M. Nykänen. Safety, translation and evaluation of alignment calculus. In *ABDIS'97*, pages 295–304.

[19] G. Grahne, M. Nykänen, E. Ukkonen. Reasoning about strings in databases. *JCSS* 59 (1999), 116–162.

[20] G. Grahne, E. Waller. How to make SQL stand for string query language. In *DBPL'99*.

[21] P. Gulutzan and S. Pelzer. *SQL-99 Complete, Really.* R&D Books, 1999.

[22] R. Hakli, M. Nykänen, H. Tamm, and E. Ukkonen. Implementing a declarative string query language with string restructuring. In *PADL'99*, pages 179–195.

[23] O. Ibarra, J. Su. A technique for proving decidability of containment and equivalence of linear constraint queries. *JCSS* 59 (1999), 1–28.

[24] N. Immerman. *Descriptive Complexity.* Springer, 1999.

[25] Ph. Kolaitis and M. Vardi. Fixpoint logic vs. infinitary logic in finite-model theory. In *LICS'92*, pages 46–57.

[26] A. Rajasekar. String-oriented databases. *SPIRE/CRIWG'99*, pages 158–167.

[27] A. Salomaa. *Formal Languages.* Academic Press, 1973.

[28] A. Stolboushkin, M. Taitslin. Finite queries do not have effective syntax. *Information and Computation*, 153(1) (1999), 99–116.