An Intensional Expressiveness Gap of Comprehension Syntax

Limsoon Wong
School of Computing, National University of Singapore, Singapore

Abstract

Comprehension syntax is widely adopted in modern programming languages as a means for manipulating collection types. This paper proves that all subquadratic algorithms which are expressible in comprehension syntax, do not compute low-selectivity joins. As database systems support these joins efficiently, this confirms an intensional expressiveness gap between comprehension syntax and relational database systems. The proof of this intensional expressiveness gap relies on a "limited-mixing" lemma which states that subquadratic algorithms expressible using comprehension syntax have limited ability for mixing atomic objects in their inputs.

2012 ACM Subject Classification Information systems → Relational database query languages

Keywords and phrases Comprehension syntax, intensional expressive power, limited-mixing lemma

Digital Object Identifier 10.4230/OASIcs.Tannen.2024.11

Funding *Limsoon Wong*: This work was supported in part by a Singapore Ministry of Education tier-2 grant, MOE-T2EP20221-0005; and in part by a Kwan Im Thong Hood Cho Temple Chair Professorship.

Acknowledgements During Val Tannen's visit to Singapore in 2019, we developed the concept of synchrony iterators [17]. Synchrony iterators enables efficient algorithms for synchronized iterations on multiple collections to be expressed elegantly using comprehension syntax. Notably, they simplify the creation of efficient algorithms for low-selectivity non-equijoins. During our discussions, I was struck by an apparent gap: the difference between the efficiency of database queries in comprehension syntax and their real-world implementation in relational database systems. I am grateful for Val's friendship and mentorship over the past three and a half decades. I am honoured and glad to contribute this chapter, which bridges the above gap in our understanding of comprehension syntax, as a tribute to Val in this festschrift volume.

1 Overview

Query languages based on comprehension syntax are able to express all relational queries supported by typical database systems [4, 22]. Moreover, queries written in comprehension syntax are appealingly simple [3]. So, comprehension syntax has become widely regarded as a means for embedding collection-type querying capabilities into programming languages. However, join queries expressed in comprehension syntax in these programming languages are generally compiled into nested loops. This implies such queries typically have quadratic or even higher time complexity when they are expressed by means of comprehension syntax.

In contrast, in relational database systems, even in the absence of indices, when joins have low selectivity, these joins often have $O(n \log n)$ time complexity based on (sort-)merge-join algorithms [2]. And when the input relations are pre-sorted on their join attributes in a low-selectivity join, a merge-join can even be realised with linear time complexity by skipping the sorting steps.

Therefore, there is a potential intensional expressiveness gap between algorithms that can be realised by comprehension syntax and those used in database systems, such as algorithms for low-selectivity joins. Consequently, despite the syntactic naturalness of comprehension syntax, one might say it fails as a genuine naturally embedded query language. Nonetheless, this gap has not been formally proven.

The main objective of this paper is to prove that this intensional expressiveness gap indeed exists. The proof goes via a "limited-mixing" lemma on $\mathcal{NRC}_1(\leq)$. On ordered data types, $\mathcal{NRC}_1(\leq)$ is equivalent to the flat relational algebra or first-order logic [22]. More pertinently, there is a simple translation between comprehension syntax and $\mathcal{NRC}_1(\leq)$, and this translation preserves time complexity. This makes $\mathcal{NRC}_1(\leq)$ a suitable ambient query language for investigating the potential intensional expressiveness gap between comprehension syntax and typical database systems.

The limited-mixing lemma states that all $\mathcal{NRC}_1(\leq)$ queries of subquadratic time complexity are only able to mix atoms in their input relations in very limited ways. So, these subquadratic-complexity queries cannot be low-selectivity joins. This limited-mixing lemma is non-query specific and is applicable even when ordered data types are present. It thus considerably enriches the available theoretical tools for studying intensional expressive power, as these tools are often query specific and are inapplicable in the presence of ordered data types. It is also a useful intensional counterpart to Gaifman's locality property [8]. Gaifman's locality is very useful for analyzing extensional expressiveness of first-order query languages on unordered data types, but is inapplicable to ordered data types.

This chapter is organized as follows. Section 2 presents $\mathcal{NRC}_1$, its operational semantics, rewrite rules and the induced normal forms. Section 3 states and proves the limited-mixing lemma. Section 4 leverages the limited-mixing lemma to prove the main result that all implementations of *zip*, which is a prototypical linear-time low-selectivity join, in $\mathcal{NRC}_1$ have at least quadratic time complexity. This confirms the intensional expressive power gap between comprehension syntax and relational database systems. Finally, Section 5 provides discussion on the intensional expressiveness gap and how the gap could be addressed.

2 Nested Relational Calculus

The restriction of the nested relational calculus $\mathcal{NRC}$ from Buneman et al. [4] and Wong [22] to flat relations is used as the ambient language here. $\mathcal{NRC}$ is equivalent to the usual nested relational algebra [4, 22]. Its restriction to flat relations, denoted here as $\mathcal{NRC}_1$, is equivalent to flat relational algebra and first-order logic [22]. This ambient language, and its operational semantics and rewrite rules, are described below.

2.1 Types and expressions

The types and expressions of $\mathcal{NRC}$ are given in Figure 1. The type superscripts in the figure are omitted when there is no confusion. For simplicity, all variable names are assumed to be distinct. For convenience, all data types are endowed with an order; this query language is denoted as $\mathcal{NRC}(\leq)$.

The semantics of a type is just a set of objects built up by nesting sets and records of base-type objects. Base types are denoted by b (representing atomic values in a database). An object of type $s_1 \times \cdots \times s_n$ is a tuple (i.e., a record) whose ith component is an object of type s_i, for $1 \leq i \leq n$. An object of type $\{s\}$ is a finite set whose elements are objects of type s; an object of type $\{s\}$ is called a set or a "relation." Moreover, if $s = b \times \cdots \times b$, then an object of type $\{s\}$ (or s) is called a "flat relation." However, if s contains some set brackets, then an object of type $\{s\}$ is called a "nested relation."

Types in $\mathcal{NRC}$

$$s ::= b \mid s_1 \times \cdots \times s_n \mid \{s\}$$
where b is a base type.

Expressions in $\mathcal{NRC}$

$$\frac{}{C^s : s} \qquad \frac{}{x^s : s} \qquad \frac{e_1 : s_1 \quad \ldots \quad e_n : s_n}{(e_1, \ldots, e_n) : s_1 \times \cdots \times s_n} \qquad \frac{e : s_1 \times \cdots \times s_n}{e.\pi_i : s_i} 1 \leq i \leq n$$

$$\frac{}{\{\}^s : \{s\}} \qquad \frac{e : s}{\{e\} : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \{s\}}{e_1 \cup e_2 : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \{t\}}{\bigcup\{e_1 \mid x^t \in e_2\} : \{s\}}$$

$$\frac{}{\mathit{true} : \mathbb{B}} \qquad \frac{}{\mathit{false} : \mathbb{B}} \qquad \frac{e_1 : \mathbb{B} \quad e_2 : s \quad e_3 : s}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : s}$$

$$\frac{e_1 : s \quad e_2 : s}{e_1 < e_2 : \mathbb{B}} \qquad \frac{e_1 : s \quad e_2 : s}{e_1 = e_2 : \mathbb{B}} \qquad \frac{e : \{s\}}{e\ \mathit{isempty} : \mathbb{B}}$$

Figure 1 $\mathcal{NRC}$.

The expression constructs are defined as follows. The expression C denotes objects, including constants of base types b; the syntax for C will be given in the next subsection. The expression $(e_1, \ldots, e_n)$ forms a tuple whose ith component is the object denoted by e_i, for $1 \leq i \leq n$. The expression $e.\pi_i$ extracts the ith component of the tuple e. The expressions $\{\}$, $\{e\}$, and $e_1 \cup e_2$ have their conventional meaning as set operations. The expression $\bigcup\{e_1 \mid x \in e_2\}$ forms the set obtained by first applying the function $f(x) = e_1$ to each object in the set e_2 and then taking their union; that is, $\bigcup\{e_1 \mid x \in e_2\} = f(C_1) \cup \ldots \cup f(C_n)$, where $f(x) = e_1$ and $\{C_1, \ldots, C_n\}$ is the set denoted by e_2.

Besides the object types and their expression constructs above, $\mathcal{NRC}$ also has the Boolean type $\mathbb{B}$ as a base type, and the expression constructs *true*, *false*, and *if* e_1 *then* e_2 *else* e_3, which have their conventional meaning as Boolean values and conditional expression. Lastly, the expression $e_1 < e_2$ provides a linear ordering on objects of the same type; the expression $e_1 = e_2$ checks whether the objects denoted by e_1 and e_2 are the same; and the expression e *isempty* checks whether the set denoted by e is empty.

The emptiness test e *isempty*, the equality test $e_1 = e_2$, and the ordering test $e_1 < e_2$ are provided for every type s solely for convenience. They are actually defined in terms of the tests on base types b. In particular, the linear ordering on any arbitrarily deeply nested combinations of record and set types can be lifted – in a manner definable by $\mathcal{NRC}$ – from the linear ordering on each base type b as follows [13]: for tuple types $s_1 \times \cdots \times s_n$, it is defined pointwise lexicographically; and for set types $\{s\}$, it is defined a la Wechler [21] based on the Hoare ordering (viz. $X \leq Y$ iff for all $x \in X - Y$, there is $y \in Y - X$, such that $x \leq y$).

The notation $x \in e_2$ in the $\bigcup\{e_1 \mid x \in e_2\}$ construct is an abstraction that introduces the variable x whose scope is the expression e_1. That is, it is part of the syntax and is not a membership test. This construct is the sole means in $\mathcal{NRC}$ for iterating over a set.

If a variable appearing in an expression e is not introduced by a subexpression of the form $\bigcup\{e_1 \mid x \in e_2\}$ in e, it is called a free variable of e. When it is necessary to explicitly indicate the free variables of an expression, we write $e(x_1, ..., x_2)$ or $e(\vec{x})$. An expression $e(\vec{x})$, with free variables $\vec{x}$ can be regarded as a function $f(\vec{x}) = e(\vec{x})$. When it is desirable to distinguish the free variables local to a subexpression $e(\vec{x}, \vec{X})$ of an expression $e'(\vec{X})$, uppercase is used for the free variables of the entire expression while lowercase is used for other free variables of the subexpression. Also, an expression e having no free variable is called a closed expression.

When objects $\vec{C}$ have the same types as the free variables $\vec{x}$ of an expression $e(\vec{x})$, the expression obtained by replacing each variable x_i in $\vec{x}$ in $e(\vec{x})$ by the corresponding C_i in $\vec{C}$ is denoted as $e[\vec{C}/\vec{x}]$. The result of applying $e(\vec{x})$ as a function to $\vec{C}$ is denoted by $e(\vec{C})$. To make notations lighter, $e(\vec{C})$ is sometimes also used to denote the expression $e[\vec{C}/\vec{x}]$; however, this usage is generally eschewed in proofs.

A "pattern-matching" construct $\bigcup\{e_1 \mid (x_1, \ldots, x_n) \in e_2\}$ is used for convenience. It is a syntactic sugar for $\bigcup\{e_1[x.\pi_1/x_1, \ldots, x.\pi_n/x_n] \mid x \in e_2\}$. There is also an easy mechanical translation [3, 22] between the syntax of $\mathcal{NRC}$ and comprehension syntax of the form $\{e \mid \delta_1, \ldots, \delta_n\}$ where each δ_i either has the form $\vec{x}_i \in e_i$ or the form e_i. The translation is as follows:

- $\{e \mid \vec{x}_1 \in e_1, \Delta\} =_{df} \bigcup\{\{e \mid \Delta\} \mid \vec{x}_1 \in e_1\}$;
- $\{e \mid e_1, \Delta\} =_{df}$ *if* e_1 *then* $\{e \mid \Delta\}$ *else* $\{\}$; and
- $\{e \mid \} =_{df} \{e\}$.

Comprehension syntax is used here to write examples, but the reader should understand these examples as syntactic sugars of the actual $\mathcal{NRC}$ expressions.

▶ **Example 1.** All relational queries [5] are expressible in $\mathcal{NRC}$.

- $\Pi_i\ X =_{df} \{x.\pi_i \mid x \in X\}$ is the relational projection;
- $\sigma_d\ X =_{df} \{x \mid x \in X,\ d(x)\}$ is the relational selection;
- $X \bowtie Y =_{df} \{(x, y) \mid (u, x) \in X,\ (v, y) \in Y,\ u = v\}$ is the relational join;
- $X \cap Y =_{df} \{x \mid x \in X,\ \textit{not}\ \{y \mid y \in Y,\ y = x\}\ \textit{isempty}\}$ is the relational intersection.
- $X - Y =_{df} \{x \mid x \in X,\ \{y \mid y \in Y,\ y = x\}\ \textit{isempty}\}$ is the relational difference; and
- $X \div Y =_{df} \{x \mid (x, y) \in X,\ Y \subseteq \{y' | (x', y') \in X,\ x' = x\}\}$, where $Y \subseteq Y' =_{df} Y - Y'\ \textit{isempty}$, is the relational division.

▶ **Example 2.** $\mathcal{NRC}$ can also express nested relational operations [20].

- *unnest* $R =_{df} \{(x, y) |\ (X, y) \in R, x \in X\}$ unnests the nested relation R; and
- *nest* $R =_{df} \{(\{x \mid (x, y) \in R, y = v\}, v) \mid (u, v) \in R\}$ creates a nested version of a relation R, which groups values in the first column of R by values in the second column of R.

Let $\mathcal{NRC}_1$ denote the fragment of $\mathcal{NRC}$ where expressions are restricted to flat relation types. That is, in $\mathcal{NRC}_1$, every (sub)expression $e(x_1, ..., x_n) : s$ where $x_i : s_i$ for $1 \leq i \leq n$, the types s, s_1, ..., s_n are all flat relations. It is known that $\mathcal{NRC}$ enjoys the conservative extension property [22]; thus, $\mathcal{NRC}(\leq)$ and $\mathcal{NRC}_1(\leq)$ express the same functions on flat relations, and are equivalent to flat relational algebra or first-order logic with ordering $FO(\leq)$.

▶ **Proposition 3.** *$\mathcal{NRC}(\leq)$, $\mathcal{NRC}_1(\leq)$, and $FO(\leq)$ have the same extensional expressive power on flat relations.*

An expression $e(\vec{x})$ in $\mathcal{NRC}$ can always be turned into an expression $e'(\vec{y}, \vec{x})$ such that no constants or objects appear in it. This can be obtained by introducing fresh free variables $\vec{y}$ and replacing each object C_i in $e(\vec{x})$ by the variable y_i; then $e'[\vec{C}/\vec{y}](\vec{x}) = e(\vec{x})$. So, for simplicity, and without loss of generality, only constant-free expressions are considered when results are stated and proved in this paper.

$$\frac{}{C \Downarrow C}$$

$$\frac{e_1 \Downarrow C_1 \quad \dots \quad e_n \Downarrow C_n}{(e_1, \dots, e_n) \Downarrow (C_1, \dots, C_n)} \qquad \frac{e \Downarrow (C_1, \dots, C_n)}{e.\pi_i \Downarrow C_i} 1 \leq i \leq n$$

$$\frac{}{\{\} \Downarrow \{\}} \qquad \frac{e \Downarrow C}{\{e\} \Downarrow \{C\}} \qquad \frac{e_1 \Downarrow C_1 \quad e_2 \Downarrow C_2}{e_1 \cup e_2 \Downarrow C_1 \oplus C_2}$$

$$\frac{\begin{array}{c} e_2 \Downarrow \{C_1, \dots, C_n\} \\ e_1[C_1/x] \Downarrow C'_1 \quad \cdots \quad e_1[C_n/x] \Downarrow C'_n \end{array}}{\bigcup\{e_1 \mid x \in e_2\} \Downarrow C'_1 \oplus \cdots \oplus C'_n}$$

$$\frac{}{\mathit{true} \Downarrow \mathit{true}} \qquad \frac{}{\mathit{false} \Downarrow \mathit{false}}$$

$$\frac{e_1 \Downarrow \mathit{true} \quad e_2 \Downarrow C}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow C} \qquad \frac{e_1 \Downarrow \mathit{false} \quad e_3 \Downarrow C}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow C}$$

$$\frac{e_1 \Downarrow C_1 \quad e_2 \Downarrow C_2}{e_1 < e_2 \Downarrow \mathit{true}} C_1 < C_2 \qquad \frac{e_1 \Downarrow C_1 \quad e_2 \Downarrow C_2}{e_1 < e_2 \Downarrow \mathit{false}} C_1 \not< C_2$$

$$\frac{e_1 \Downarrow C_1 \quad e_2 \Downarrow C_2}{e_1 = e_2 \Downarrow \mathit{true}} C_1 = C_2 \qquad \frac{e_1 \Downarrow C_1 \quad e_2 \Downarrow C_2}{e_1 = e_2 \Downarrow \mathit{false}} C_1 \neq C_2$$

$$\frac{e \Downarrow C}{e\ \mathit{isempty} \Downarrow \mathit{true}} C = \{\} \qquad \frac{e \Downarrow C}{e\ \mathit{isempty} \Downarrow \mathit{false}} C \neq \{\}$$

Figure 2 A call-by-value operational semantics of $\mathcal{NRC}$.

2.2 Operational semantics

In order to discuss intensional expressive power, i.e. what algorithms are expressible, it is necessary to know how an expression of $\mathcal{NRC}$ is executed. This is specified in Figure 2 as a call-by-value operational semantics. A call-by-value operational semantics is widely adopted in programming languages and has also been used for several variations of $\mathcal{NRC}$ in earlier works [18, 19, 24] on intensional expressive power.

In Figure 2, the notation $e \Downarrow C$ means the closed expression e is evaluated to produce the object C. The unique evaluation tree of e is denoted using the notation $e \Downarrow$. The "step" complexity $\mathit{step}(e \Downarrow)$ of an evaluation is defined as the time complexity of the largest node in the evaluation tree – viz., $\mathit{step}(e \Downarrow) = \max\{\mathit{time}(e' \Downarrow C') \mid$ the node $e' \Downarrow C'$ occurs in the evaluation tree of $e \Downarrow$. The time complexity $\mathit{time}(e' \Downarrow C')$ of a node is the number of branches that the node has. E.g., in Figure 2, $\mathit{time}(\bigcup\{e_1 \mid x \in e_2\} \Downarrow C'_1 \oplus \cdots \oplus C'_n) = n + 1$. On the other hand, the time complexity $\mathit{time}(e \Downarrow)$ of an evaluation is the sum of the time complexity of all the nodes in the tree.

The syntax for objects C is as follows. A constant c of a base type b is an object of type b. A tuple $(C_1, \ldots, C_n)$ is an object of type $s_1 \times \cdots \times s_n$ if each C_i is an object of type s_i. An "enumeration list", elist for short, $\{C_1, \ldots, C_n\}$ is an object of type $\{s\}$ if each C_i is an object of type s. An elist $\{C_1, .., C_n\}$ can be thought of as a particular way of enumerating the elements of the set that it represents, viz. C_1 followed by C_2, followed by C_3, and so on. There are as many distinct elists that represent the same set as there are distinct ways to enumerate elements of that set, corresponding to different ordering and multiplicity of appearances of its elements in the enumeration.

The notations $C = C'$ and $C == C'$ are used to refer to two notions of equality involving elists. The notation $C = C'$ means C are C' are the same objects when all the elists contained in them (and objects therein) are interpreted as sets: thus, $c = c'$ iff c and c' are the same constant of a base type; $(C_1, ..., C_n) = (C'_1, ..., C'_n)$ iff $C_i = C'_i$ for $1 \leq i \leq n$; and $\{C_1, ..., C_n\} = \{C'_1, ..., C'_m\}$ iff for each $1 \leq i \leq n$, there is $1 \leq j \leq m$ such that $C_i = C'_j$, and for each $1 \leq j \leq m$, there is $1 \leq i \leq n$ such that $C_i = C'_j$. The notation $C == C'$ means C and C' are the same objects when all the elists contained in them (and objects therein) are interpreted as lists: thus, $c == c'$ iff c and c' are the same constant of a base type; $(C_1, ..., C_n) == (C'_1, ..., C'_n)$ iff $C_i == Ci'$ for $1 \leq i \leq n$; and $\{C_1, ..., C_n\} == \{C'_1, ..., C'_m\}$ iff $n = m$, and $C_i == C'_i$ for $1 \leq i \leq n$.

In Figure 2, a constructor $C \oplus C'$ is used to produce the concatenation of two elists in constant time; i.e. given $C == \{C_1, ..., C_n\}$ and $C' == \{C'_1, ..., C'_m\}$, $C \oplus C' == \{C_1, ..., C_n, C'_1, ..., C'_m\}$. Also, $\oplus$ is always used in a right-associative manner; e.g., $C \oplus C' \oplus C''$ means $C \oplus (C' \oplus C'')$. Note that while it is not a common practice to use a constant-time concatenation constructor to represent lists, it has been used in e.g. the influential Kleisli Query System [23] which is based on $\mathcal{NRC}$.

Linear orderings $<$ are available on all base types and are lifted to all types, as defined earlier. With this, the subset of objects in "canonical form" can be defined as follows. A constant c of any base type b is canonical. A tuple $(C_1, ..., C_n)$ is canonical if each C_i is canonical. An elist $\{C_1, ..., C_n\}$ is canonical if for every $1 \leq i, j \leq n$, it is the case that C_i is canonical, C_j is canonical, and $C_i < C_j$ iff $i < j$; a canonical elist is thus duplicate-free and is sorted according to $<$. The notation $\mathit{canonize}(C)$ denotes the unique canonical form of the object C. Clearly, for $C == \{C_1, ..., C_n\}$ representing a flat relation, $\mathit{canonize}(C)$ can be produced in $O(n \log(n))$ time.

The call-by-value operational semantics in Figure 2 does not perform canonization. This is because canonization is not needed to guarantee the soundness of an evaluation in $\mathcal{NRC}(\leq)$.

▶ **Proposition 4** (Soundness). *Suppose $e(\vec{x}) : s$ is an expression in $\mathcal{NRC}$, $\vec{C}$ are objects having the same types as $\vec{x}$, and $e[\vec{C}/\vec{x}] \Downarrow C'$. Then $e[\vec{C}/\vec{x}] = C'$.*

The size of an object C can be defined in any reasonable way. One way is defining $\mathit{size}(C)$ as the number of symbols used to write C out. Another way, when C is an elist, is defining $\mathit{size}(C)$ as $|C|$, the length of the elist. Both notions of size can be generalized to $\mathit{size}(\vec{C}) = \sum_i \mathit{size}(C_i)$. The latter notion of input size is used by default. Then the time complexity of an expression $e(\vec{x})$ can be defined in the usual way based on input size; i.e. the time complexity of $e(\vec{x})$ is a function $g : \mathbb{N} \to \mathbb{N}$ where $g(n)$ equals the maximum of $\mathit{time}(e[\vec{C}/\vec{x}] \Downarrow C')$ over all inputs $\vec{C}$ of size at most n. Then, the time complexity is said to be constant if g is $\Theta(1)$, linear if g is $\Theta(n)$, quadratic if g is $\Theta(n^2)$, and polynomial if g is $\Theta(n^k)$ for some natural number k. The following is easily shown in a manner similar to [4, Theorem 4.4].

▶ **Proposition 5** (Polynomiality). *Let $e(\vec{x}) : s$ be an expression in $\mathcal{NRC}(\leq)$. Then there is a number k such that the time complexity of $e(\vec{x})$ is $\Theta(n^k)$ where n denotes input size. In particular, if the time complexity of $e(\vec{x})$ is sub-quadratic, then it must be either linear or constant time; and if it is sub-linear, then it must be constant time. Furthermore, these properties are retained when $\mathcal{NRC}$ is augmented by any additional functions that have polynomial time complexity.*

$$
\begin{array}{rcl}
\bigcup\{e \mid x \in \{\}\} & \mapsto & \{\} \\
\bigcup\{e_1 \mid x \in \{e_2\}\} & \mapsto & e_1[e_2/x] \\
\bigcup\{e \mid x \in (e_1 \cup e_2)\} & \mapsto & \bigcup\{e \mid x \in e_1\} \cup \bigcup\{e \mid x \in e_2\} \\
\bigcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\} & \mapsto & \bigcup\{\bigcup\{e_1 \mid x \in e_2\} \mid y \in e_3\} \\
\bigcup\{e \mid x \in (\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3)\} & \mapsto & \mathit{if}\ e_1\ \mathit{then}\ \bigcup\{e \mid x \in e_2\}\ \mathit{else}\ \bigcup\{e \mid x \in e_3\} \\
(e_1, \ldots, e_2).\pi_i & \mapsto & e_i \\
(\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3).\pi_i & \mapsto & \mathit{if}\ e_1\ \mathit{then}\ e_2.\pi_i\ \mathit{else}\ e_3.\pi_i \\
\mathit{if}\ \mathit{true}\ \mathit{then}\ e_2\ \mathit{else}\ e_3 & \mapsto & e_2 \\
\mathit{if}\ \mathit{false}\ \mathit{then}\ e_2\ \mathit{else}\ e_3 & \mapsto & e_3
\end{array}
$$

Figure 3 A system of rewrite rules for $\mathcal{NRC}$.

2.3 Rewrite rules

Figure 3 shows a system of rewrite rules for simplifying $\mathcal{NRC}$ expressions. These rules have been used in many previous works on $\mathcal{NRC}$ [22, 14, 15, 24]. These rules are easily shown to be sound, and do not increase step complexity, and are strongly normalizing [22].

Although this system of rewrite rules does not increase step complexity, it can increase time complexity. E.g., rewriting $\bigcup\{\bigcup\{\{(x, z)\} \mid z \in Z\} \mid x \in \{\bigcup\{\{y.\pi_1\} \mid y \in Y\}\}\}$ to $\bigcup\{\{(\bigcup\{\{y.\pi_1\} \mid y \in Y\}, z)\} \mid z \in Z\}$ by the second rule in Figure 3, changes the time complexity from $O(|Y| + |Z|)$ to $O(|Z| \cdot |Y|)$.

Fortunately, the second rule in Figure 3 is the only rule that misbehaves this way. For convenience of reference, the system of rewrite rules in Figure 3 is called the unrestricted system. And when the second rule is excluded, it is called the restricted system.

▶ **Proposition 6** (Normal form). *Let $e(\vec{X}) : s$ be an expression in $\mathcal{NRC}(\leq)$, and $\vec{C}$ be objects having the same types as $\vec{X}$.*

1. *$e[\vec{C}/\vec{X}] == e'[\vec{C}/\vec{X}]$ if $e \mapsto e'$.*
2. *$\mathit{step}(e[\vec{C}/\vec{X}] \Downarrow) \geq \mathit{step}(e'[\vec{C}/\vec{X}] \Downarrow)$ if $e \mapsto e'$.*
3. *$\mathit{time}(e[\vec{C}/\vec{X}] \Downarrow) \geq \mathit{time}(e'[\vec{C}/\vec{X}] \Downarrow)$ if $e \mapsto e'$ under the restricted system of rewrite rules.*
4. *The (un)restricted system of rewrite rules is strongly normalizing.*
5. *The unrestricted system of rewrite rules induces a normal form, wherein every subexpression of the form $\bigcup\{e_1(y, \vec{x}, \vec{X}) \mid y \in e_2(\vec{x}, \vec{X})\}$, $e_2(\vec{x}, \vec{X})$ must be one of the variables in $\vec{X}$.*
6. *The restricted system of rewrite rules induces a normal form, wherein every subexpression of the form $\bigcup\{e_1(y, \vec{x}, \vec{X}) \mid y \in e_2(\vec{x}, \vec{X})\}$, $e_2(\vec{x}, \vec{X})$ must be one of the variables in $\vec{X}$ or $e_2(\vec{x}, \vec{X})$ has the form $\{e_3(\vec{x}, \vec{X})\}$.*

3 A limited-mixing lemma

An analysis of the normal form induced by the restricted system of rewrite rules yields a useful limited-mixing lemma on $\mathcal{NRC}_1(\leq)$. The lemma is proved below, after some relevant definitions are given.

A level-0 atom of an object C is a constant c which has at least one occurrence in C that is not inside any elist in C. A level-1 atom of an object C is a constant c which has at least one occurrence in C that is inside an elist which is not nested inside another elist in C. All other constants appearing in an object C are higher level atoms. The notations $\mathit{atom}^0(C)$, $\mathit{atom}^1(C)$, and $\mathit{atom}^{\leq 1}(C)$ respectively denote the set of level-0 atoms of C, the set of level-1 atoms of C, and their union. The level-0 molecules of an object C are the elists in C that are not nested inside other elists. The notation $\mathit{molecule}^0(C)$ denotes the set of level-0 molecules of C. E.g., suppose $C = (c_1, c_2, \{(c_3, c_4, \{(c_5, c_6)\})\})$; then $\mathit{atom}^0(C) = \{c_1, c_2\}$, $\mathit{atom}^1(C) = \{c_3, c_4\}$, $\mathit{atom}^{\leq 1}(C) = \{c_1, c_2, c_3, c_4\}$, $\{c_5, c_6\}$ are higher-level atoms, and $\mathit{molecule}^0(C) = \{\{(c_3, c_4, \{(c_5, c_6)\})\}\}$.

The level-0 Gaifman graph of an object C is defined as an undirected graph $\mathit{gaifman}^0(C)$ whose nodes are the level-0 atoms of C, and edges are all the pairs of level-0 atoms of C. The level-1 Gaifman graph of an object C is defined as an undirected graph $\mathit{gaifman}^1(C)$ whose nodes are the level-1 atoms of C, and the edges are defined as follow: If $C == \{C_1$, ..., $C_n\}$, the edges are pairs (x, y) such that x and y are in the same $\mathit{atom}^0(C_i)$ for some $1 \leq i \leq n$; if $C == (C_1$, ..., $C_n)$, the edges are pairs $(x, y) \in \mathit{gaifman}^1(C_i)$ for some $1 \leq i \leq n$; and there are no other edges. The Gaifman graph [8] of an object C is defined as $\mathit{gaifman}(C) = \mathit{gaifman}^0(C) \cup \mathit{gaifman}^1(C)$.

It is shown below, by induction on the structure of $\mathcal{NRC}_1(\leq)$ expressions, that they manipulate their inputs in highly restricted local manners. In particular, expressions which have contant time complexity are unable to mix level-0 and level-1 atoms. And expressions which have linear time complexity are able to mix level-0 atoms with level-0 and level-1 atoms, but are unable to mix level-1 atoms with themselves or with higher-level atoms.

▶ **Lemma 7** (Limited mixing). *Let $e(\vec{X}) : s$ be an expression in $\mathcal{NRC}_1(\leq)$. Suppose objects $\vec{C}$ have the same types as $\vec{X}$, and $e[\vec{C}/\vec{X}] \Downarrow C'$.*

1. *If $e(\vec{X})$ has constant time complexity, then*
   (i) $\mathit{atom}^0(C') \subseteq \mathit{atom}^0(\vec{C})$,
   (ii) $\mathit{atom}^1(C') \subseteq \mathit{atom}^{\leq 1}(\vec{C})$,
   (iii) $\mathit{gaifman}(C') \subseteq \mathit{gaifman}(\vec{C})$,
   (iv) for each $U \in \mathit{molecule}^0(C')$, there are V_0, V_1, ..., V_m such that $\mathit{atom}^1(V_0) \subseteq \mathit{atom}^0(\vec{C})$, $V_j \in \mathit{molecule}^0(\vec{C})$ for each $1 \leq j \leq m$, and $U = V_0 \cup V_1 \cup \cdots \cup V_m$.
2. *If $e(\vec{X})$ has linear time complexity, then*
   (i) $\mathit{atom}^0(C') \subseteq \mathit{atom}^0(\vec{C})$,
   (ii) $\mathit{atom}^1(C') \subseteq \mathit{atom}^{\leq 1}(\vec{C})$, and
   (iii) for each $(u, v) \in \mathit{gaifman}(C')$, either $(u, v) \in \mathit{gaifman}(\vec{C})$, or $u \in \mathit{atom}^0(\vec{C})$ and $v \in \mathit{atom}^1(\vec{C})$, or $u \in \mathit{atom}^1(\vec{C})$ and $v \in \mathit{atom}^0(\vec{C})$.

Proof. The proof proceeds by structural induction on $e(\vec{X})$. For Part 1, the only interesting case is when $e(\vec{X})$ has constant time complexity and has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in e_2(\vec{X})\}$. This implies both $e_1(x, \vec{X})$ and $e_2(\vec{X})$ have constant time complexity. Let $\vec{C}$ have the types of $\vec{X}$, $e[\vec{C}/\vec{X}] \Downarrow C'$, and $e_2[\vec{C}/\vec{X}] \Downarrow C''$. Then by induction hypothesis on $e_2(\vec{X})$, $\mathit{atom}^0(C'') \subseteq \mathit{atom}^0(\vec{C})$, $\mathit{atom}^1(C'') \subseteq \mathit{atom}^{\leq 1}(\vec{C})$, and $\mathit{gaifman}(C'') \subseteq \mathit{gaifman}(\vec{C})$. Note that $\mathit{molecule}^0(C'') = \{C''\}$. The induction hypothesis implies $C'' = V_0 \cup V_1 \cup \cdots \cup V_m$ where $\mathit{atom}^1(V_0) \subseteq \mathit{atom}^0(\vec{C})$ and $V_j \in \mathit{molecule}^0(\vec{C})$ for each $j > 0$. This means each V_j, $j > 0$, is one of the input relations in $\vec{C}$. However, this leads to a contradiction because $\bigcup\{e_1(x, \vec{X}) \mid x \in e_2(\vec{X})\}$ would then have at least linear time complexity. So, there can be no V_j, $j > 0$. Hence, $C'' = V_0$ and $\mathit{atom}^1(C'') = \mathit{atom}^1(V_0) \subseteq \mathit{atom}^0(\vec{C})$. Let $C'' = \{C_1, \ldots, C_n\}$. Then, $\mathit{atom}^0(C_i) \subseteq \mathit{atom}^0(\vec{C})$. Let $e_1[C_i/x, \vec{C}/\vec{X}] \Downarrow C_i'$. Then, by induction hypothesis on $e_1(x, \vec{X})$, $\mathit{atom}^0(C_i') = \{\} \subseteq \mathit{atom}^0(C_i, \vec{C}) = \mathit{atom}^0(\vec{C})$, $\mathit{atom}^1(C_i') \subseteq \mathit{atom}^{\leq 1}(\vec{C})$, $\mathit{gaifman}(C_i') \subseteq \mathit{gaifman}(C_i, \vec{C}) = \mathit{gaifman}(\vec{C})$. The induction hypothesis also implies $C_i' = V_{i,0} \cup V_{i,1} \cup \cdots \cup V_{i,m}$ for some m where $\mathit{atom}^0(V_{i,0}) \subseteq \mathit{atom}^0(C_i, \vec{C}) = \mathit{atom}^0(\vec{C})$, and $V_{i,j} = \mathit{molecule}^0(C_i, \vec{C}) = \mathit{molecule}^0(\vec{C})$ for $j > 0$. Thus, each C'_i satisfies Part 1(i) to Part 1(iv). Consequently, $C' = C'_1 \cup \cdots \cup C'_n$ satisfies Part 1(i) to Part 1(iv). The other cases for Part 1 are straightforward, and are thus omitted.

For Part 2, by Proposition 6, $e(\vec{X})$ is assumed to be in the normal form induced by the restricted system of rewrite rules. This first interesting case is when $e(\vec{X})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in X_0\}$, where X_0 is one of the free variables in $\vec{X}$, and has linear time complexity. Then $e_1(x, \vec{X})$ must have constant time complexity; otherwise, the whole expression has quadratic time complexity. Let $\vec{C}$ have the types of $\vec{X}$ and let C_0 in $\vec{C}$ correspond to X_0. Suppose $C_x \in C_0$ and $e_1[C_x/x, \vec{C}/\vec{X}] \Downarrow C'_x$. As C'_x has set type, $\mathit{atom}^0(C'_x) = \{\} \subseteq \mathit{atom}^0(\vec{C})$. This proves Part 2(i). Since $C_x \in C_0$ and C_0 is in $\vec{C}$, $\mathit{atom}^0(C_x) \in \mathit{atom}^1(\vec{C})$. Also, as this lemma concerns $\mathcal{NRC}_1$, C_x must have type $b \times \cdots \times b$; thus, $\mathit{atom}^1(C_x) = \{\}$. By the induction hypothesis on $e_1(x, \vec{X})$, $\mathit{atom}^1(C'_x) \subseteq \mathit{atom}^{\leq 1}(C_x, \vec{C}) = \mathit{atom}^{\leq 1}(\vec{C})$. This proves Part 2(ii). As $e_1(x, \vec{X})$ has constant time complexity, and C_x has type $b \times \cdots \times b$, the induction hypothesis also implies $\mathit{gaifman}(C'_x) \subseteq \mathit{gaifman}(C_x, \vec{C}) = \mathit{gaifman}^0(C_x, \vec{C}) \cup \mathit{gaifman}^1(\vec{C})$. Suppose $(u, v) \in \mathit{gaifman}(C'_x)$. If $u \in \mathit{atom}^0(C_x)$ and $v \in \mathit{atom}^0(C_x)$, then $(u, v) \in \mathit{gaifman}^1(\vec{C}) \subseteq \mathit{gaifman}(\vec{C})$. If $u \in \mathit{atom}^0(C_x)$ and $v \notin \mathit{atom}^0(C_x)$, then $u \in \mathit{atom}^1(C_0) \subseteq \mathit{atom}^1(\vec{C})$ and $v \in \mathit{atom}^0(\vec{C})$. If $u \notin \mathit{atom}^0(C_x)$ and $v \in \mathit{atom}^0(C_x)$, then $v \in \mathit{atom}^1(C_0) \subseteq \mathit{atom}^1(\vec{C})$ and $u \in \mathit{atom}^0(\vec{C})$. If $u \notin \mathit{atom}^0(C_x)$ and $v \notin \mathit{atom}^0(C_x)$, then both u and v are in $\mathit{atom}^0(\vec{C})$, and thus $(u, v) \in \mathit{gaifman}^0(\vec{C}) \subseteq \mathit{gaifman}(\vec{C})$. This proves Part 2(iii). This finishes the case when $e(\vec{X})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in X_0\}$,

The second interesting case is when $e(\vec{X})$has linear time complexity and has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in \{e_2(\vec{X})\}\}$. Let $\vec{C}$ have the types of $\vec{X}$ and let $e_2[\vec{C}/\vec{X}] \Downarrow C''$. By the induction hypothesis of either Part 1 or 2 (it does not matter which), we get $\mathit{atom}^0(C'') \subseteq \mathit{atom}^0(\vec{C})$; thus, $\mathit{atom}^0(C'', \vec{C}) = \mathit{atom}^0(\vec{C})$. Since $\{e_2(\vec{X})\}$ has flat relation type, $e_2(\vec{X})$ must have a type of the form $b \times \cdots \times b$. This means $\mathit{atom}^1(C'') = \{\} \subseteq \mathit{atom}^1(\vec{C})$; thus, $\mathit{atom}^1(C'', \vec{C}) = \mathit{atom}^1(\vec{C})$. Crucially, $\mathit{atom}^0(C'') \subseteq \mathit{atom}^0(\vec{C})$ and $\mathit{atom}^1(C'') = \{\}$ implies $\mathit{gaifman}(C'', \vec{C}) = \mathit{gaifman}(\vec{C})$. As C'' has no elist, $\mathit{molecule}^0(C'', \vec{C}) = \mathit{molecule}^0(\vec{C})$. Then both Part 1 and 2 of the lemma follows immediately for this case.

The other cases are straightfoward and are omitted. ◀

4 Intensional expressiveness gap

As mentioned earlier, an intensional expressiveness gap of comprehension syntax relative to relational database systems appears to manifest in joins of low selectivity. And judging by Example 1, it also potentially manifests in relational intersection and relational difference, as these two operations have $O(n \log n)$ time complexity in a relational database system whereas their comprehension-syntax equivalent in Example 1 is quadratic. The other relational query operations (project, select, and union), as well as joins of high selectivity are succinctly expressible in $\mathcal{NRC}_1(\leq)$ with comparable time complexity when there are no indices available on the input relations; cf. Example 1. The relational division is ignored here because it is not directly supported by typical relational database systems; i.e., when it is needed in a relational database system, it is expressed using the other operators, usually at quadratic space and time complexity [11].

This intensional expressiveness gap is illustrated and confirmed here using two example queries on objects in canonical form. The first query, $\mathit{head}(x, X)$, produces the first element in an input canonical elist X, assuming this first element has the form (x, x') and x does not appear in subsequent elements of X. The second query, $\mathit{zip}(X, Y)$, produces an elist that pairs the ith elements in two input canonical elists X and Y of equal length, assuming the

ith element of X has the form (o_i, x'_i) and that in Y has the form (o_i, y'_i) and that each o_i occurs only once in X and once in Y. These two queries are chosen because *head* can be straightforwardly implemented in constant time in any programming language, while *zip* is a very low-selectivity join which can be answered efficiently – i.e. with linear or near-linear time complexity – in relational database systems.

The expression $\mathit{head}'(x, X) =_{df} \{(y, y') \mid (y, y') \in X, y = x\}$ in $\mathcal{NRC}_1(\leq)$ defines the same function as *head* on any input (x, X) meeting the requirement of *head*. However, $\mathit{head}'(x, X)$ has time complexity $\Theta(|X|)$; i.e., it has linear time complexity.

The expression $\mathit{zip}'(X, Y) =_{df} \{(x, y) \mid (u, x) \in X, (v, y) \in Y, u = v\}$ in $\mathcal{NRC}_1(\leq)$ defines the same function as *zip* on any input (X, Y) meeting the requirement of *zip*. However, $\mathit{zip}'(X, Y)$ has time complexity $\Theta(|X| \cdot |Y|)$; i.e., it has quadratic time complexity.

In fact, as shown below, every expression in $\mathcal{NRC}_1(\leq)$ that implements *head* has at least linear time complexity; and every expression in $\mathcal{NRC}_1(\leq)$ that implements *zip* has at least quadratic time complexity. In other words, the intensional expressiveness gap of $\mathcal{NRC}_1(\leq)$, and thus of comprehension syntax, is real.

▶ **Proposition 8.** *Let* $\mathit{head}(x, X) : \{b_1 \times b_2\}$ *be an expression in* $\mathcal{NRC}_1(\leq)$*. Suppose for every object* c *of type* b_1 *and non-empty canonical object* C *of type* $\{b_1 \times b_2\}$ *whose first element is* (c, c_0)*, and* c *does not appear in subsequent elements of* C*,* $\mathit{head}[c/x, C/X] \Downarrow \{(c, c_0)\}$*. Then* $\mathit{time}(\mathit{head}[c/x, C/X] \Downarrow)$ *is at least* $|C|$*. That is, the time complexity of* $\mathit{head}(x, X)$ *is* $\Omega(|X|)$*.*

Proof. For a contradiction, suppose $\mathit{head}(x, X)$ has sublinear time complexity. Then Proposition 5 implies $\mathit{head}(x, X)$ has constant time complexity. Let $\mathit{head}[c/x, C/X] \Downarrow C'$ where $C' = \{(c, c_0)\}$. As C' has type $\{b_1 \times b_2\}$, $\mathit{molecule}^0(C') = \{C'\}$. Similarly, $\mathit{molecule}^0(c, C) = \{C\}$. By Part 1(iv) of Lemma 7, either $C \subseteq C'$ or $\mathit{atom}^1(C') \subseteq \mathit{atom}^0(c, C)$. However, $C \not\subseteq C' = \{(c, c_0)\}$ in general and $\mathit{atom}^1(C') = \{c, c_0\} \not\subseteq \mathit{atom}^0(c, C) = \{c\}$. This contradiction implies that $\mathit{head}(x, X)$ has at least linear time complexity. ◀

▶ **Proposition 9.** *Let* $\mathit{zip}(X, Y) : \{b_1 \times b_2\}$ *be an expression in* $\mathcal{NRC}_1(\leq)$ *where* X *is a variable of type* $\{b_3 \times b_1\}$*,* Y *is a variable of type* $\{b_3 \times b_2\}$*, and* b_1*,* b_2*, and* b_3 *are distinct base types. Suppose for every canonical objects* $U == \{(o_1, u_1), ..., (o_n, u_n)\}$ *of type* $\{b_3 \times b_1\}$ *and* $V == \{(o_1, v_1), ..., (o_n, v_n)\}$ *of type* $\{b_3 \times b_2\}$*,* $\mathit{zip}[U/X, V/Y] \Downarrow C'$ *where* $C' == \{(u_1, v_1), ..., (u_n, v_n)\}$*. Then* $\mathit{time}(\mathit{zip}[U/X, V/Y] \Downarrow)$ *is at least* $|U| \cdot |V|$*. Thus, the time complexity of* $\mathit{zip}(X, Y)$ *is* $\Omega(|U| * |Y|)$*.*

Proof. Suppose for a contradiction that $\mathit{zip}(X, Y)$ has subquadratic time complexity. Then Proposition 5 implies $\mathit{zip}(X, Y)$ has either constant or linear time complexity.

Assume $\mathit{zip}(X, Y)$ has constant time complexity and $\mathit{zip}[U/X, V/Y] \Downarrow C'$ where $C' == \{(u_1, v_1), ..., (u_n, v_n)\}$. Clearly, $\mathit{molecule}^0(C') = \{C'\}$ and $\mathit{molecule}^0(U, V) = \{U, V\}$. Then, by Part 1(iv) of Lemma 7, either $U \subseteq C'$, $V \subseteq C'$, or $\mathit{atom}^1(C') \subseteq \mathit{atom}^0(U, V) = \{\}$. Clearly, all three options are impossible. Thus $\mathit{zip}(X, Y)$ cannot have constant time complexity.

Suppose instead $\mathit{zip}(X, Y)$ has linear time complexity. Then $\mathit{gaifman}(C') = C' = \{(u_1, v_1), ..., (u_n, v_n)\}$. However, for $1 \leq i \leq n$, $(u_i, v_i) \in \mathit{gaifman}(C') \notin \mathit{gaifman}(U, V) = U \cup V$. Then, by Part 2(iii) of Lemma 7, either $u_i \in \mathit{atom}^0(U, V)$ or $v_i \in \mathit{atom}^0(U, V)$. However, as U and V are both elists, $\mathit{atom}^0(U, V) = \{\}$ and thus contains neither u_i nor v_i. So, $\mathit{zip}(X, Y)$ cannot have linear time complexity. Therefore, it has at least quadratic time complexity. ◀

Incidentally, it was Peter Buneman who first conjectured that *zip*, characterized by its low-selectivity nature, could only be defined using comprehension syntax with quadratic time complexity. This insight, shared with me by Stijn Vansummeren over three decades ago, has apparently remained open until being resolved here in Proposition 9.

5 Closing remarks

The impedance mismatch problem between databases and programming languages has been highlighted three decades ago [7]. It refers to the difficulties of integrating database query-like feature and capability into a programming language. Some has regarded the use of comprehension syntax [3] as a breakthrough for this problem [6]. Indeed, comprehension syntax provides an iteration construct that is simple enough for programming with collection data types that data objects of a database have been mapped to, and explicit enough to admit a direct translation to the query language of the database, thereby permitting queries to the database to be embedded simply and naturally into a programming language.

However, comprehension syntax is also widely adopted in modern programming languages – e.g., Python [9] and Scala [16] – as an easy-to-use means for manipulating collection types in general. For this purpose, the collection objects are created within a program or do not come from a database system, and queries written in comprehension syntax for manipulating these objects are not translated to the query language of an underlying database system for execution. In such a setting, programs written in comprehension syntax typically correspond to nested loops.

This gives rise to an intriguing disparity. Many queries when translated to their database equivalent can be executed by the underlying database system very efficiently. Yet when they are executed directly as comprehension syntax, they are not efficient at all. Consider this query as an example, $\{(x.\mathit{dept}, x.\mathit{stf}) \mid x \in \mathit{DeptStaff},\ y \in \mathit{Staff},\ x.\mathit{stf} = y.\mathit{stf},\ y.\mathit{age} > 65\}$ which retrieves departments and their staff who are above 65 years old. Suppose a staff typically belongs to only one department. This query would then be a low-selectivity join. It typically would be executed by a database system, via e.g. a merge join [2], with time complexity $\Theta(n+m)$ assuming the inputs *DeptStaff* and *Staff* have size n and m and are both sorted by their *stf* field; or with time complexity $\Theta(n \log(n) + m \log(m))$ if sorting is required. In contrast, the same query would typically has time complexity $\Theta(nm)$ natively in the programming language. Even if a filter promotion is applied (and ignoring the change in the appearance of the output) to optimize the query to $\{(x.\mathit{dept}, x.\mathit{stf}) \mid y \in \mathit{Staff},\ y.\mathit{age} > 65,$ $x \in \mathit{DeptStaff},\ x.\mathit{stf} = y.\mathit{stf}\}$, this optimized query still has quadratic time complexity $\Theta(gnm)$, for some $0 \leq g \leq 1$, natively in the programming language.

This linear-vs-quadratic time complexity difference of low-selectivity joins can be called an intensional expressiveness gap between comprehension syntax and database systems. That is, it is a gap between the algorithms that can be expressed using comprehension syntax and database systems. As far as relational database system is concerned, the low-selectivity join, the relational intersection, and the relational difference appear to be the only intensional expressiveness gap as all other relational query operators, as well as high-selectivity joins, in the absence of database indices on the input relations, have similar time complexity whether executed by a relational database system or in the programming language directly as queries in comprehension syntax.

It has been open whether this intensional expressiveness gap is a real gap; i.e., there might exist some clever way to implement low-selectivity joins efficiently using comprehension syntax. As the main result of this paper, this intensional expressiveness gap is proved by showing that all subquadratic algorithms expressible using pure comprehension syntax cannot compute low-selectivity joins. In fact, I have claimed elsewhere [17] that even allowing some functions – viz. *takewhile* and *dropwhile*, *fold*, or *zip* – commonly available in the collection-type function libraries of programming languages, to be used with comprehension syntax, all expressible subquadratic algorithms still cannot compute low-selectivity joins in general.

It is a natural follow-up question on what exactly is missing from comprehension syntax that prevents efficient algorithms for low-selectivity joins to be expressed. This intensional expressiveness gap can be charaterized in a precise way by identifying a new programming construct that enables more algorithms to be expressed but doing so without enabling more functions to be expressed. This is the Synchrony iterator construct, which I have proposed and investigated with Val Tannen and Stefano Perna [17], for expressing synchronized iterations on multiple collection objects. A construct for generalized iteration on multiple collection objects in synchrony appears to be a conceptually novel choice, because practically all functions commonly provided in the function libraries of programming languages involve iteration on a single collection object. Adding this construct does not change the functions that are expressible using pure comprehension syntax, and yet enables the realization of efficient low-selectivity joins, including non-equijoins. Moreover, the Synchrony iterator construct dovetails rather appealingly with comprehension syntax, so that efficient queries written with the help of Synchrony iterators often do not look too different from their inefficient pure comprehension-syntax equivalents. See [17] for more information.

The proof of the intensional expressiveness gap uses a novel limited-mixing lemma. The lemma shows that all subquadratic-time queries in comprehension syntax are only able to mix atomic objects in their input in very limited ways. This limited-mixing lemma is of independent interest. Many past works on intensional expressive power are query specific. Just to cite a couple of examples, Abiteboul and Vianu [1] showed that there is no "generic machine" for computing the parity query in PTIME; and Suciu and Paredaens [18] showed that the transitive closure of a long chain can only be computed in the complex object algebra of Abiteboul and Beeri using exponential space. A notable non-query-specific intensional expressiveness result is that of Wong [24], who showed that all queries on a general class of structures, which includes deep trees and long chains, in a nested relational calculus augmented with a powerset operator are either already expressible in the calculus without using the powerset operator, or must use an exponential amount of space. Furthermore, most previous results on intensional expressive power, such as those mentioned above, are for query languages without ordered data types. The limited-mixing lemma in this paper stands out in comparison to these results in two aspects. Firstly, the limited-mixing lemma is non-query specific; it applies to all queries of subquaratic time complexity in the respective query languages. Secondly, the limited-mixing lemma is valid in the presence of ordered data types. The limited-mixing lemma thus enriches the repertoire of techniques for studying intensional expressive power. The limited-mixing lemma is also useful intensional counterpart to Gaifman's locality property [8]. Gaifman's locality property is useful for analyzing the extensional and intensional expressive power [10, 12, 24] of query languages on unordered data types. However, it is effectively useless on ordered data types and on query languages with a *fold*-like function. Limited-mixing lemmas do not have these limitations.

Lastly, here is a small advertisement: Synchrony iterator has been implemented in Python and Scala. These implementations are available at `https://www.comp.nus.edu.sg/~wongls/projects/synchrony`.

References

1 Serge Abiteboul and Victor Vianu. Generic computation and its complexity. In *Proceedings of 23rd ACM Symposium on the Theory of Computing*, pages 209–219, 1991.

2 Michael W. Blasgen and Kapali Eswaran. Storage and access in relational databases. *IBM Systems Journal*, 16(4):363–377, 1977.

3 Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

4 Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995.

5 Edgar F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice-Hall, 1972.

6 Ezra Cooper. The script-writer dream: How to write great SQL in your own language, and be sure it will succeed. In *Proceedings of 12th International Symposium on Database Query Languages*, pages 36–51, Lyon, France, August 2009.

7 George Copeland and David Maier. Making Smalltalk a database system. In *Proceedings of ACM-SIGMOD 84*, pages 316–325, Boston, MA, June 1984.

8 Haim Gaifman. On local and non-local properties. In *Proceedings of the Herbrand Symposium, Logic Colloquium '81*, pages 105–135. North Holland, 1982.

9 John V. Guttag. *Introduction to Computation and Programming Using Python: With Application to Understanding Data*. MIT Press, 2016.

10 Lauri Hella, Leonid Libkin, and Juha Nurmonen. Notions of locality and their logical characterizations over finite models. *Journal of Symbolic Logic*, 64(4):1751–1773, 1999.

11 Dirk Leinders and Jan Van den Bussche. On the complexity of division and set joins in the relational algebra. *Journal of Computer and System Sciences*, 73(4):538–549, June 2007.

12 Leonid Libkin. On the forms of locality over finite models. In *Proceedings of 12th IEEE Symposium on Logic in Computer Science*, pages 204–215, 1997.

13 Leonid Libkin and Limsoon Wong. Aggregate functions, conservative extension, and linear orders. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 282–294. Springer-Verlag, January 1994.

14 Leonid Libkin and Limsoon Wong. Conservativity of nested relational calculi with internal generic functions. *Information Processing Letters*, 49(6):273–280, March 1994.

15 Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *Journal of Computer and System Sciences*, 55(2):241–272, October 1997.

16 Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Inc., December 2019.

17 Stefano Perna, Val Tannen, and Limsoon Wong. Iterating on multiple collections in synchrony. *Journal of Functional Programming*, 32:e9, July 2022.

18 Dan Suciu and Jan Paredaens. The complexity of the evaluation of complex algebra expressions. *Journal of Computer and Systems Sciences*, 55(2):322–343, October 1997.

19 Dan Suciu and Limsoon Wong. On two forms of structural recursion. In *LNCS 893: Proceedings of 5th International Conference on Database Theory*, pages 111–124, Prague, January 1995. Springer-Verlag.

20 Stan J. Thomas and Patrick C. Fischer. *Nested Relational Structures*, pages 269–307. JAI Press, London, England, 1986.

21 Wolfgang Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monograph on Theoretical Computer Science*. Springer-Verlag, Berlin, 1992.

22 Limsoon Wong. Normal forms and conservative extension properties for query languages over collection types. *Journal of Computer and System Sciences*, 52(3):495–505, June 1996.

23 Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, 2000.

24 Limsoon Wong. A dichotomy in the intensional expressive power of nested relational calculi augmented with aggregate functions and a powerset operator. In *Proceedings of 32nd ACM Symposium on Principles of Database Systems*, pages 285–295, New York, June 2013.