Database Research needs an Abstract Relational Query Language

Wolfgang Gatterbauer
Northeastern University, USA

Diandre Miguel Sabale
Northeastern University, USA

ABSTRACT

For decades, SQL has been the default language for composing queries, but it is increasingly used as an artifact to be read and verified rather than authored. With Large Language Models (LLMs), queries are increasingly machine-generated, while humans read, validate, and debug them. This shift turns relational query languages into interfaces for back-and-forth *communication about intent*, which will lead to a rethinking of relational language design, and more broadly, relational interface design.

We argue that this rethinking needs support from an *Abstract Relational Query Language* (ARQL): a semantics-first reference metalanguage that separates query intent from user-facing syntax and makes underlying relational patterns explicit and comparable across user-facing languages. An ARQL separates a query into (*i*) a *relational core* (the compositional structure that determines intent), (*ii*) *modalities* (alternative representations of that core tailored to different audiences), and (*iii*) conventions (orthogonal environment-level semantic parameters under which the core is interpreted, e.g., set vs. bag semantics, or treatment of null values). Usability for humans or machines then depends less on choosing a particular language and more on choosing an appropriate modality. Comparing languages becomes a question of which relational patterns they support and what conventions they choose.

We introduce Abstract Relational Calculus (ARC), a strict generalization of Tuple Relational Calculus (TRC), as a concrete instance of ARQL. ARC comes in three modalities: (*i*) a comprehension-style textual notation, (*ii*) an Abstract Language Tree (ALT) for machine reasoning about meaning, and (*iii*) a diagrammatic hierarchical-graph (higraph) representation for humans. ARC provides the missing vocabulary and acts as a Rosetta Stone for relational querying.

KEYWORDS

Relational Language Design, Query Understanding

☞ This PDF contains internal hyperlinks for easier reading: click any linked term to jump to the section where it is defined.

1 INTRODUCTION

New interfaces for humans and machines. Several recent efforts question SQL as the default relational query language (QL) and propose to either extend or completely replace it [8, 46, 52]. Examples in these debates include whether nested correlated queries are inherently hard for users to follow and should be replaced with more dataflow (algebraic) abstractions, and whether set or bag semantics are the right choice (e.g., debated at the DBPL workshop at SIGMOD'25 [34]). Less explored, however, is a more basic representational question: ❶ *How should we represent the intent of a query so that its relational structure is explicit and comparable across different surface syntaxes? How can we describe how a query composes its inputs (i.e., the base relations) to define query intent, independent of the idiosyncrasies of any particular query language?*

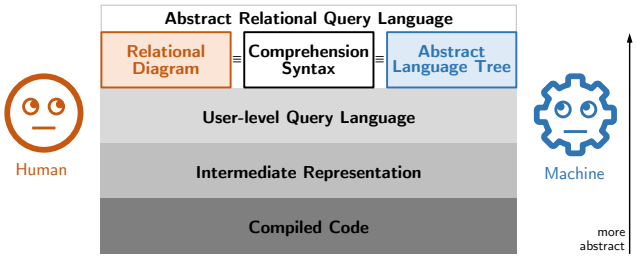


Figure 1: An *Abstract Relational QL* (ARQL) abstracts away from syntactic details of a query to a higher-level representation. Just as Intermediate Representations (IRs) enable query optimization, a more abstract representation can support semantic understanding of a query's intent. Both humans and machines can benefit from *modalities* tailored to their needs. *Conventions* (not shown) factor out orthogonal design choices that don't affect the relational pattern.

At the same time, relational queries are increasingly produced by machines and validated by humans. In this setting, the user interaction with relational databases changes: now SQL is not just a language for users composing queries, but it also increasingly serves as a message format between machine generation and human validation. As noted in the Cambridge report [7], these developments "*potentially change how we interface with relational databases*," shifting emphasis from query composition to query interpretation. "*The essential skill is no longer simply writing programs but learning to read, understand, critique and improve them instead*" [51]. Since LLMs can hallucinate or introduce errors, "*effective explanation mechanisms ... become increasingly important*" [7]. This raises a second question: ❷ *How should machine-generated queries be presented to users so they can validate them and provide feedback?*

The challenge is not limited to human-facing interfaces. Machine-facing tasks such as semantic similarity search and retrieval also require representations aligned with meaning rather than syntax. SQL's surface syntax is a poor proxy for intent: semantically equivalent queries can differ substantially in syntactic structure, while syntactically similar queries may encode different semantics. In the NL2SQL domain, current benchmarks often rely on surface-level criteria such as exact string match or execution match. Since those fail to capture deeper semantic relationships, Floratou et al. [22] argue for "*a shift towards intent-based benchmarking frameworks.*" This raises a third question: ❸ *What language abstraction should an LLM (or future machine-tool) use to internally reason about query intent and semantic similarity in a way that is faithful to relational meaning?*

Our Suggestion. We believe that database research needs *new vocabulary* to analyze relational intent across languages and a

 16th Annual Conference on Innovative Data Systems Research (CIDR '26). January 18-21, Chaminade, USA

semantics-focused representation of relational languages that decouples intent from user-facing syntax while supporting multiple modalities (i.e., mechanically inter-translatable representations of the same language tailored to different audiences, without treating each modality as a separate language). We call such a representation an *Abstract Relational Query Language* (ARQL).

It is *abstract* because it factors out differences in concrete syntax and design choices and instead focuses on a small set of compositional relational primitives shared across relational query languages. In this sense, a representation is *more abstract* when it makes the relational intent of a query [14, 25] explicit without relying on syntactic shortcuts or being forced to expose "conventions" in the language. In general, we refer to a language-agnostic description of how data is transformed from input to output as the *relational pattern* of a query [30]. Our goal is a clean *separation of concerns*. Just as Intermediate Representations (IRs), such as SDQL [50] and Substrait [5], decouple front-end parsing from optimization and code generation, we want to enable a *syntax-agnostic discussion of language features at a more conceptual level*. This lets us treat relational patterns as modules (Section 2.13.2) and several issues that are not necessarily part of the relational pattern of a query as orthogonal choices ("conventions", see Section 2.6), such as the convention of using set or bag semantics, the blurry distinction between declarative and procedural languages, the treatment of null values, typing and casting conventions, as well as different initializations of aggregate functions (e.g., 0 or null for `sum`). It also allows us to discuss the many syntactic variants that SQL permits for expressing basically the same intent, as well as when rewrites are not equivalent (e.g., the COUNT bug, see Section 3.2).

Another concrete use case is NL2SQL. Rather than generating SQL text directly, an NL2SQL system can generate an ARQL representation of intent and then render it into SQL. In this paper, our concrete ARQL instance is Abstract Relational Calculus (ARC) (introduced in Section 2) with a machine-facing Abstract Language Tree (ALT) modality, which provides a natural intermediate target for NL2SQL systems.

Modalities instead of languages. An ARQL does not have just one representation. Instead, we propose developing alternative *modalities* of the same language, each tailored to different purposes. These modalities offer alternative views of a query, targeted for either human interpretation or machine reasoning.

Concretely, what are usually called *Abstract Syntax Trees* (ASTs) tend to remain too close to the concrete syntax of a language. For example, SQLGlot's AST [4, 56] places JOINs as children of SELECT nodes, which reflects surface-level parsing (concrete syntax) rather than abstract semantic relationships. We argue that such representations fall short as truly abstract representations of a query.

We use the term *Abstract Language Tree* (ALT) for a universal, hierarchically structured representation of the *semantics of a query* rather than its syntax. We originally used the term *Abstract Language Higraph* (ALH) instead of ALT. The motivation is that ASTs and ALTs are "trees" only with respect to the containment (nesting) structure (and even that can be blurred by correlated constructs such as `LATERAL` joins). Once name resolution is performed and bindings are established (i.e., identifier occurrences are connected to their declarations via cross-references, and the resulting structure is often called an *annotated/decorated AST* [13]), the overall

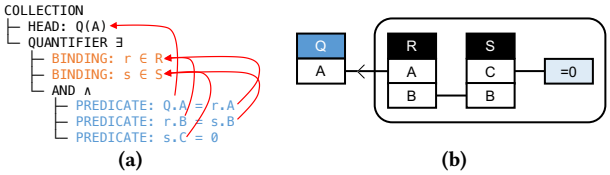


Figure 2: (a): Linked Abstract Language Tree (ALT) for TRC (1). The overlaid arrows show the result of the linking step and are conceptual only. (b): Diagrammatic higraph representation of the linked ALT as a variant of Relational Diagrams.

structure is better viewed as a hierarchical graph (a tree of containment with additional edges). This also matches the intuition that lexical scopes correspond to nested regions (Fig. 2b). *Higraphs* [36] formalize exactly this combination of nesting and linking: nodes may be nested within nodes (capturing containment/scopes) while edges capture references. However, they are not widely known, and the term is unfamiliar to many readers (for a simplified and accessible higraph formalism, see the online appendix of [28]). Thus, we ultimately kept the simpler term Abstract Language Tree (ALT): the conceptual shift from surface syntax to the underlying semantic operations is already substantial, while the remaining intuitions from ASTs carry over, and our higraph modality continues to make the hierarchical graph (higraph) data structure explicit.

Similar to how query graphs support the optimization of conjunctive queries [44], we believe ALTs provide a better data structure for semantic analysis of relational queries. For an ARQL, the language-independent ALT is ideally identical to its AST, because the syntax reflects its semantics.

Importantly, ALTs can also be rendered diagrammatically for human users as hierarchical graphs (higraphs). In that form, the nested scopes of nodes in the ALT are replaced by a nesting of nodes. Prior user studies have shown that Relational Diagrams can help users understand relational structures faster and more reliably [29, 41]. This emphasizes a key distinction: *language design* should not be conflated with *interface usability*. Whether users "like" a language is a question of modality, not of the language core itself. Instead, modalities should be designed with target consumers in mind, i.e. human-facing modalities for accurate semantic understanding and debugging, and machine-facing modalities for tasks such as semantic similarity assessment or query transformation. Thus, an ARQL provides the relational structure, while modalities are lossless presentations of that structure for different consumers.

While we agree with the observation that "*the idea of a single, universal language or paradigm ... covering all data programming needs is unlikely*" [7], we argue that many of these needs can be addressed at the level of modalities instead of languages. The goal of an ARQL is not to unify all QLs under a single syntax, but to enable meaningful comparisons across languages in terms of their underlying relational patterns, and different modalities can serve the respective needs of humans and machines. The translation between modalities can also be automated.

Conventions instead of languages. In Green's cognitive dimensions of notations [32] a "system" consists of notation (the representational form) and an environment (the surrounding tooling). Similarly, we suggest distinguishing between *a language* (a

representation that encodes the relational composition of a query in a particular surface syntax) and a *convention* (an orthogonal design decision that can be switched and will affect the behavior but not the relational core). For example, the aggregate `sum(R.A)` initializes with null in SQL, but with 0 in Soufflé (Section 2.6). This difference is a design decision. It is a convention that does not affect the way a relational query composes its various components to encode a meaning. An ARQL focuses only on the relational patterns of a query and does not expose conventions which are specified separately in the surrounding environment. With this change, a sufficiently generic language design could be interpreted under either set or bag semantics. It is just a switch that we flip on or off. While discussion of set vs. bag semantics is still important for query optimization, it becomes orthogonal to "*language design*."

Contributions. ❶ We suggest that the database community develops abstract representations of relational queries that can embed relational query patterns across user-facing relational query languages. This effort can support, but is orthogonal to, the development of concrete user-level Query Languages (QLs) and efforts on Intermediate Representations (IRs) (Section 1). ❷ We propose a concrete formulation of such a language called Abstract Relational Calculus (ARC), which is a strict generalization of Tuple Relational Calculus (TRC) that assumes flat relational inputs and outputs and so far has 3 modalities. By making implicit relational constructs and dependencies explicit, ARC abstracts and surfaces common query patterns found across different relational languages in a more explicit representation. By treating human- and machine-facing representations as modalities of the same underlying calculus, it supports a more principled discussion of relational language design, for both the future human and machine audiences (Section 2). ❸ We show ARC representations of running examples from recent and older papers, which we believe support an ongoing discussion (Section 3 and examples interspersed throughout Section 2).

2 ABSTRACT RELATIONAL CALCULUS (ARC)

We formalize Abstract Relational Calculus (ARC), a strict generalization of Tuple Relational Calculus (TRC) that models relational query languages in a collection framework.[1]

ARC is an Abstract Relational Query Language (ARQL): a semantics-first reference metalanguage that can encode the core relational query patterns of SQL and various proposed alternatives. We present ARC in 3 modalities: (*i*) a comprehension-based syntax that generalizes TRC, (*ii*) an Abstract Language Tree (ALT) suited for machine reasoning, and (*iii*) a diagrammatic higraph modality suited for human inspection. Although equivalent, each modality is tailored to a different audience.

2.1 Starting with TRC

We start with TRC because we have a strong conviction that the named calculus perspective is a more suitable abstraction for an ARQL than positional addressing. Codd [11] proposed to "*replace positional addressing by totally associative addressing*", i.e. accessing values by named attributes rather than by argument position. This gives us logical independence not only from tuple order (row position), but also from attribute order (column positions). Moreover, Boolean statements in TRC are always domain independent [28] as long as all range variables are bound to relations, a property that is not widely known and does not hold for DRC.

Several recent works in our community are inspired by Datalog, due to its handling of recursion. However, nothing prevents us from adding recursion in the named attribute perspective (Section 2.9). The positional (domain) perspective is also favored for conciseness: One can simply write $R(x, y)$ for predicate or function application instead of $\exists r \in R\,[r.A = x \wedge r.B = y]$. For an ARQL, however, brevity is diametrically opposed to its goal of surfacing patterns across languages, and making otherwise implicit constructs explicit. As one example, an assignment predicate $Q.A = r.A$ in TRC, $\{Q(A) \mid \exists r \in R\,[Q.A = r.A]\}$, has no explicit counterpart in DRC, where the same binding is implicit in the output tuple $\{(x) \mid R(x)\}$. Conciseness is often associated with usability (fewer letters to type). But we associate usability primarily with the chosen modality rather than the relational core (see Fig. 2a vs. Fig. 2b, which render the same language in two different modalities; the diagrammatic modality supports faster human inspection).

Ignoring notational conventions, the following is a valid TRC query according to a widely used textbook [20]:

$$\{r.A \mid r \in R \wedge \exists s[r.B = s.B \wedge s.C = 0 \wedge s \in S]\}$$

We make two changes. First, we clarify the scopes. Whenever a relation variable is quantified, then it is also bound to a relation:

$$\{r.A \mid r \in R, \exists s \in S[r.B = s.B \wedge s.C = 0]\}$$

Second, we have stricter scoping rules. *We do not allow variables bound in the body to appear in the head.* Instead, we assign values to the head variables explicitly via an assignment predicate:

$$\{Q(A) \mid \exists r \in R, s \in S[Q.A = r.A \wedge r.B = s.B \wedge s.C = 0]\} \quad (1)$$

This means that all bindings (e.g., $s \in S$) are now introduced by an explicit quantifier. Notice that two bindings can share the same quantifier ($\exists r \in R, s \in S$). We call the extra predicate $Q.A = r.A$ an *assignment predicate* to distinguish it from the other *comparison predicates*. This membership-style formalization of TRC is developed in great detail in [28].

2.2 Language Modalities

Abstract Language Tree (ALT). Figure 2a shows our formalism of an Abstract Language Tree (ALT) representation of (1) which makes this nesting of one or more bindings under a quantifier explicit [28]. Notice that a query (or a collection) consists of a head and a formula as body, and a quantification starts the body.

We also show conceptual links from predicates to the bindings of their range variables. These are not typically shown in ASTs, but they reflect the data structures created after the linking step and symbol tables are created. Given what we perceive as a confusion about what an *Abstract* Syntax Tree (AST) is supposed to represent (recall Section 1), we say ALT but actually think about this linked and hierarchical data structure as an Abstract Language

[1] We were considering the more explicit name *Abstract Tuple Relational Calculus* to emphasize the lineage and leave space for a possible future Abstract Domain Relational Calculus. But the more we thought about it, the more we came to believe that the domain perspective is not well-suited for an ARQL (though it may well be a suitable choice for a user-facing syntax as in Rel [8]). Alternative names considered were Generalized TRC and Extended TRC (as in extended relational algebra).

Higraph (ALH). Developing a good set of data abstractions is essential for solving problems. We believe a hierarchical graph is a good abstraction for relational structures.

Relational Diagrams (Higraph diagrams). For computational analysis, this pointer-based hierarchical graph structure is appropriate. For human consumption, we use a diagrammatic representation of the ALT where scopes represented as nodes in the ALT become regions, and where the attributes of a table are represented adjacent to the table name instead of using additional edges. For the relationally complete fragment, these concepts were already formalized as Relational Diagrams in more detail in [28–30]. A user study has shown that these diagrams allow humans to recognize and reason about patterns faster than SQL. The user study was reproduced [55]. Two recent tutorials [26, 27] give a detailed comparison of this visual formalism against prior work.

Two minor differences from that prior work are: (*i*) we now explicitly represent existential scopes (previously omitted because, under set semantics, only negation requires an unambiguous scope interpretation; this changes under bag semantics and aggregation), and (*ii*) we visually decorate assignment predicates (crucial for nested comprehensions).

2.3 Interpreting TRC as set comprehension

Everything so far was grounded in first-order logic. We next interpret relational query languages in a collection framework, viewing a query as an expression in a comprehension calculus with tuple variables, quantifiers, and scoping. This interpretation will allow us later to go beyond first-order logic, yet remain declarative.

A declarative specification of a set can be given by specifying the elements that satisfy the properties of the set. For example, for sets X and Y, the subset of their product where the former is smaller than the latter is the set $\{(x, y) \mid x < y \wedge x \in X \wedge y \in Y\}$. With our stricter scoping rules, we would write it instead as

$$\{(a, b) \mid \exists x \in X, y \in Y[x < y \wedge a = x \wedge b = y]\}$$

In a Haskell list comprehension syntax, this would be written as[2]

```
[(x,y) | x <- xs, y <- ys, x < y]
```

with its semantics given by the conceptual evaluation strategy:

```
for x in X:
  for y in Y:
    if x < y: yield (x, y)
```

This nested loop strategy gives both a semantic and operational definition and is exactly the way we also explain the conceptual evaluation strategy of SQL in our undergraduate database courses. In the following, we use this formalism of comprehension of sets (and more generally collections) yet also deviate in three details: 1) We use a tuple instead of a domain perspective. 2) We have an explicit notation of scoping: The body can be a logical statement instead of a conjunction of properties, thus the order of shown predicates does not matter. What matters are the well-defined scopes. 3) We use our stricter rule on heads (heads need to be kept clean).

Notice that allowing nesting in the head is usually *the way* of defining calculations with collections and list comprehensions. For example, in Haskell creating all the squares of even numbers is canonically written with squaring of numbers happening in the head:

```
[x*x | x<-ns, mod x 2 == 0]
```

Nesting in the head is also useful for the nested relational model, and used in extensions of Datalog. But nesting in the head is not needed (Section 2.12) and we believe it is distracting for the flat (unnested) relational model.

[2]Since value variables must start lowercase in Haskell, we use xs and ys instead of our otherwise preferred notation X and Y.

```
select x.A, z.B
from X as x
join lateral (
  select y.A as B
  from Y as y
  where x.A < y.A) as z
on true
```

(a)

Figure 3: (a): Nested ARC from (2) expressed as lateral join in SQL.

2.4 Composability through orthogonal nesting

We allow arbitrary nesting of comprehensions in the body, i.e. nesting is orthogonal (and therefore compositional): it does not interfere with or restrict other constructs (subject to scoping rules). For example, in Haskell, the following comprehension is allowed:

```
[(x, z) | x <- xs, z <- [ y | y <- ys, x < y]]
```

This expression would correspond in SQL to the lateral join shown in Fig. 3a. It is written in ARC as follows:

$$\{Q(A, B) \mid \exists x \in X, z \in \{Z(B) \mid \exists y \in Y[Z.B = y.A \wedge x.A < y.A] \\ [Q.A = x.A \wedge Q.B = z.B]\} \quad (2)$$

2.5 Grouping and aggregates in set semantics

Consider the task of summing the salaries of all employees in a table. Under set semantics, projecting the salary column *before applying an aggregate function removes duplicates*, so we obtain the sum of distinct salaries rather than the sum over all employees. This is usually not the intended behavior when computing aggregates.

A conceptually simple fix is to apply aggregate functions not over individual columns but over entire sets of tuples, with each aggregate operating on a designated position of those tuples (e.g., the last position in the unnamed perspective). This is the approach proposed by Klug [40] in his classical 1982 paper, in which each aggregate function receives *its own separate scope*.[3] Subsequent comprehension-based database programming languages (DBPLs) [9, 21, 33, 53], including extensions of logic with aggregate operators [37], and modern Datalog-inspired systems such as Soufflé [2] and Rel [8], inherited variants of this formalism.

This formalism has left two important legacies for set-based languages: (1) Evaluating two different aggregate functions over the same relation requires two logical copies of that relation because

[3]Like Datalog, Klug works in the unnamed (positional) perspective. To treat aggregates as ordinary unary function symbols on relations, each aggregate must be tied to a fixed column in advance. Consequently, applying a sum to the 2nd or the 3rd column of the same relation requires two *different* aggregate functions (the column index is effectively baked into the function name). This need for separate aggregates per column complicates the formalism notably.

each aggregate is evaluated in its own independent scope. (2) Except for Rel [8], aggregate functions in these formalisms return only their computed results, not the grouped attributes. These attributes must instead be specified *outside* the aggregation scope and passed into it via correlated nesting (a pattern we refer to as "from the outside in" = FOI).

We discuss these issues in detail after introducing ARC's handling of aggregates. Notice how ARC serves as a *reference language that provides the vocabulary and structure* needed to compare and reason about the behavior of different languages. In doing so, ARC *abstracts from the idiosyncrasies of each language's surface syntax and instead captures their shared underlying semantic structure.*

Aggregation in ARC. ARC's collection framework supports a conceptual evaluation strategy in which *aggregates are defined over the full join*:[4] values are accumulated one element at a time, and multiple aggregates can be evaluated in parallel by reusing the same scope, as in SQL. Thus, conceptually, an aggregate has two inputs: the full join (determined by the scope in which the aggregation predicate appears) and a column identifier (i.e. range variables and column name, as in TRC). If desired, deduplication of the input values can be expressed either by first applying a projection or by using dedicated aggregate functions (e.g., countdistinct instead of count). To enforce set semantics of the output, deduplication is applied to the result values of the scope. This is a flexible pattern that allows ARC to recover the different interpretations of aggregates from existing relational languages.

Recall that our aim is not to retrofit aggregates into classical first-order logic [37], but to provide an abstract calculus that can model the diverse computational patterns (including aggregation) found in real relational languages. Relational Calculus here is meant as a more general term than first-order logic. As Date writes [15, Ch. 8] "*calculus ... provides a notation for stating the definition of that desired relation in terms of those given relations*", and "*A fundamental feature of the calculus is the range variable*", and "*variable that "ranges over" some specified relation.*" This definition of calculus naturally motivates our proposal of Abstract Relational Calculus (ARC), an abstract relational query language defined in a collection framework that strictly generalizes TRC.

Consider a simple grouped aggregate query over a binary relation $R(A, B)$ that computes, for each distinct value of $R.A$, the sum of associated $R.B$ values. ARC expresses this query in comprehension syntax as follows:

$$\{Q(A, sm) \mid \exists r \in R, \gamma_{r.A}[Q.A = r.A \wedge Q.sm = \text{sum}(r.B)]\} \tag{3}$$

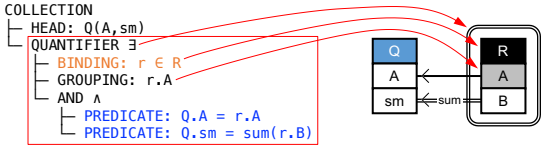
The *aggregation predicate* $Q.sm = \text{sum}(r.B)$ accumulates and aggregates values over the set of tuples produced by the conceptual evaluation strategy.[5] Thus, aggregates appear as operands in predicates. The query has a *grouping operator* γ with *grouping key* $r.A$ that partitions the full join result within the quantifier scope into groups based on bindings of the grouping key $r.A$. When the aggregate is taken over the entire join result, then we write $\gamma_\emptyset$ explicitly (similar to "group by true" in SQL). Thus, the appearance of any

[4]Recall that a full join of two or more relations contains no duplicates (i.e. it is a set) if the input relations are sets as well.

[5]Notice that the aggregation predicate serves here simultaneously also as assignment predicate, although aggregation predicates can also serve as comparison predicates as we see later.

```
select R.A, sum(R.B) sm
from R
group by R.A
```

(a) Grouped aggregate in SQL



(b) Grouped aggregate in ARC as ALT and higraph modalities

Figure 4: The semantics of a simple grouped aggregate query in a "*from the inside out*" pattern represented in SQL (a) and ARC (b), (3). Red overlay arrows indicate scoping, binding and grouping.

aggregation predicate turns an existential scope into a grouping scope and requires a grouping operator.

In the comprehension syntax and ALT, the grouping operator is a child of the quantification scope and turns this scope into a *grouping scope*. In the higraph modality, grouped attributes are highlighted with a gray shade, and the scope is drawn with a double-lined boundary to indicate a grouping scope (Fig. 4b).

From the inside out (FIO). In ARC, grouping and aggregation happen on attributes inside a scope, and the resulting grouped and aggregated attributes are then available outside that scope. We therefore call this pattern "*from the inside out.*" It corresponds exactly to the way SQL would represent the grouped aggregate query (Fig. 4a) [3, Database 720], and to extended relational algebra:

$$\gamma_{A,\text{sum}(B)\to\text{sm}}[R]$$

Rel [8] has the same relational pattern:

```
def Q(a,sm) : sm = sum[(b) : R(a,b)]
```

Rel's interpretation of aggregate queries is that of variable elimination [38], and the query as written in Rel can be viewed as creating a lookup function f with $f(a) := \sum_{b:R(a,b)} b$.

From the outside in (FOI). Several languages represent our simple grouped aggregate query in a "per-outer-tuple" pattern. Klug's formalism [40] uses an unnamed variant of a tuple relational calculus that uses nesting in the head. Slightly changing the syntax and porting it to a named perspective, the query would be written as follows:[6]

$$\{(r.A, \text{sum}_2\{(r_2.A, r_2.B) \mid r_2 \in R \wedge r_2.A = r.A\}) \mid r \in R\} \tag{4}$$

Here, the first range variable $r \in R$ fixes the grouping key $r.A$, and the second range variable $r_2 \in R$ performs the aggregation for each value in $r.A$.

Hella et al. [37] use the same pattern, but in an unnamed domain perspective:

$$(\exists y.R(x, y)) \wedge (q = \text{Aggr}_\Sigma\ z.(R(x, z), z)) \tag{5}$$

In this query, x is a free variable, and the conjunct $\exists y.R(x, y)$ range-restricts its admissible values to those that occur in $R.A$. The aggregate $\text{Aggr}_\Sigma\ z.(R(x, z), z)$ binds z and is parameterized by the free variable x, which serves as the grouping key.

[6]Original notation: $(v_1[1], \text{sum}_2((v_2[1], v_2[2]) : R(v_2) : v_2[1] = v_1[1])) : R(v_1)$.

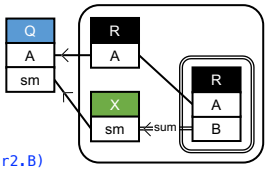
```
select distinct R.A,
  (select sum(R2.B) sm
  from R R2
  where R2.A=R.A)
from R
```

(a) Scalar subquery in SQL

```
select distinct R.A, X.sm
from R join lateral
  (select sum(R2.B) sm
  from R R2
  where R2.A=R.A) X
on true
```

(b) Lateral join in SQL

```
COLLECTION
├─ HEAD: Q(A,sm)
└─ QUANTIFIER ∃
   ├─ BINDING: r ∈ R
   ├─ BINDING: x ∈
   │  └─ COLLECTION
   │     ├─ HEAD: X(sm)
   │     └─ QUANTIFIER ∃
   │        ├─ BINDING: r2 ∈ R
   │        ├─ GROUPING: ∅
   │        └─ AND ∧
   │           ├─ PREDICATE: r2.A = r.A
   │           └─ PREDICATE: X.sm = sum(r2.B)
   └─ AND ∧
      ├─ PREDICATE: Q.A = r.A
      └─ PREDICATE: Q.sm = x.sm
```



(c) FOI pattern in ARC

Figure 5: The semantics of a simple grouped aggregate query in a "*from the outside in*" pattern represented in SQL with a scalar subquery (a) or lateral join (b), and ARC (c), (7). This relational pattern corresponds to the way Klug [40] (4), Hella et al. [37] (5), and Soufflé [2, 49] (6) express the query.

In Soufflé [2, 49], the query follows the same pattern:

```
Q(a,sum b: {R(a,b)}) :- R(a,_).                    (6)
```

The documentation [2] describes this pattern explicitly: "*You cannot export information from within the body of an aggregate. This means that you cannot ground a variable from within the scope of the aggregate body and expect this grounding to transfer to the outer scope.*"

This pattern of using two range variables over the same relation to perform a grouped aggregate query without an explicit GROUP BY clause can be represented in SQL either via scalar subqueries or via lateral joins. Figure 5a and Fig. 5b show two different syntactic variants in SQL that represent this pattern. Notice that both queries are semantically equivalent: a single-valued scalar query can always be written as a lateral join (see Section 2.12).

In ARC, the same pattern expressed as ALT, higraph, and comprehension syntax is shown in Fig. 5c and as:

$$\{Q(A, sm) \mid \exists r \in R, x \in \{X(sm) \mid \exists r_2 \in R, \gamma_\emptyset [r2.A = r.A \wedge X.\text{sm} = \text{sum}(r_2.B)]\}[Q.A = r.A \wedge Q.\text{sm} = x.\text{sm}]\} \quad (7)$$

Notice that while SQL does not use an explicit grouping clause when an aggregate is computed over the entire relation, ARC indicates the aggregate evaluation explicitly with a grouping on the empty set: there is just one group, an aggregate is evaluated over all tuples (similar to "group by true" in SQL), and ARC makes this explicit.

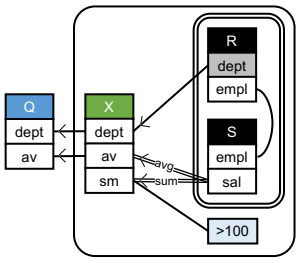
Notice that ARC introduces an explicit intermediate defined relation X that exists only implicitly in the surface syntax of several languages. While function composition, as in $(g \circ f)(x) = g(f(x))$ or the head-nested scalar subquery, hides an intermediate relation, the lateral-join formulation in SQL already exposes it as a derived relation. Recall that an abstract relational query language serves as a *reference language* and *should make implicit patterns explicit*. Thus, ARC represents these conceptual structures explicitly as defined relations, which provides a clear abstraction for understanding how queries are built in a modular way from smaller components, even when they are never or cannot be materialized (see Section 2.13). Also notice that we do not need to name them in the higraph modality (Fig. 5c): they exist on the Canvas as independent topological entities and may remain unnamed.

```
select R.dept, avg(S.sal) av
from R, S
where R.empl=S.empl
group by R.dept
having sum(S.sal)>100
```

(a) Multiple aggregates in SQL

```
COLLECTION
├─ HEAD: Q(dept,av)
└─ QUANTIFIER ∃
   ├─ BINDING: x ∈
   │  └─ COLLECTION
   │     ├─ HEAD: X(dept,av,sm)
   │     └─ QUANTIFIER ∃
   │        ├─ BINDING: r ∈ R
   │        ├─ BINDING: s ∈ S
   │        ├─ GROUPING: r.dept
   │        └─ AND ∧
   │           ├─ PREDICATE: r.empl= s.empl
   │           ├─ PREDICATE: X.dept = r.dept
   │           ├─ PREDICATE: X.av = avg(s.sal)
   │           └─ PREDICATE: X.sm = sum(s.sal)
   └─ AND ∧
      ├─ PREDICATE: Q.dept = x.dept
      ├─ PREDICATE: Q.av = x.av
      └─ PREDICATE: x.sm > 100
```

(b) ARC ALT modality



(c) ARC higraph modality

Figure 6: (a): Running example for multiple aggregates from [37] in SQL (a) and ARC (b), (c), (8).

Multiple aggregates. We illustrate our formalism using the running example from Hella et al. [37] "*returning the average salary for each department that pays total salary at least* 100" over a schema (with slightly simplified relation names and constant) $R(empl, dept), S(empl, sal)$ representing employees, their departments, and their salaries. Figure 6a shows the corresponding SQL query [3, Database 740].

In ARC, a HAVING clause is simply a selection applied after an aggregation:

$$\{Q(dept, av) \mid \exists x \in \{X(dept, av, sm) \mid \exists r \in R, s \in S, \gamma_{r.dept} [X.dept = r.dept \wedge X.av = \text{avg}(s.sal) \wedge X.sm = \text{sum}(s.sal) \wedge r.empl = s.empl]\}[Q.dept = x.dept \wedge Q.av = x.av \wedge x.sm > 100]\} \quad (8)$$

The query expressed in the language $\mathcal{L}_{\text{aggr}}(\{<\}, \{\sum, \text{AVG}\})$ by Hella et al. [37] defines an output relation $Q(y, q)$ via the following

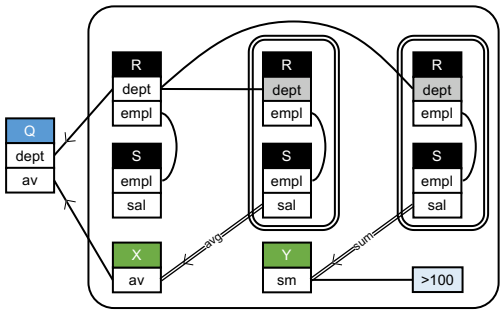


Figure 7: Pattern-preserving ARC higraph representation of the multiple-aggregate query in the formalism by Hella et al. [37] (9).

expression:[7]

$$Q(y,q) := (\exists x \exists z.R(x,y) \wedge S(x,z)) \quad (9)$$
$$\wedge\ (q = \mathrm{Aggr}_{\mathrm{AVG}} x,z.(R(x,y) \wedge S(x,z),z))$$
$$\wedge\ (\mathrm{Aggr}_{\sum} x,z.(R(x,y) \wedge S(x,z),z) > c_{100})$$

For a fixed y, the aggregate term $\mathrm{Aggr}_{\sum}\, x,z.(R(x,y) \wedge S(x,z),z)$ ranges over all *distinct* rows (x,z) such that $R(x,y) \wedge S(x,z)$ holds. It collects the bag $\{\!\{z \mid \exists x,z[R(x,y) \wedge S(x,z)]\}\!\}$ (with multiplicities) and applies the summation operator Σ (sum) to that bag. This formalism (inherited from Klug [40]) changes the signature of the query: the same base relations are referenced multiple times, once in each aggregation scope, and once outside the aggregation scopes. This leads to a *modified relational pattern*, shown in the higraph modality (Fig. 7) and in comprehension syntax modalities:

$$\{Q(dept, av) \mid \exists r_3 \in R, s_3 \in S, \quad (10)$$
$$x \in \{X(av) \mid \exists r_1 \in R, s_1 \in S, \gamma_{r_1.dept}$$
$$[r_1.dept = r_3.dept \wedge r_1.empl = s_1.empl \wedge X.av = \mathrm{avg}(s_1.sal)]\},$$
$$y \in \{Y(sm) \mid \exists r_2 \in R, s_2 \in S, \gamma_{r_2.dept}$$
$$[r_2.dept = r_3.dept \wedge r_2.empl = s_2.empl \wedge Y.sm = \mathrm{sum}(s_2.sal)]\}$$
$$[Q.dept = r_3.dept \wedge Q.av = x.av \wedge r_3.empl = s_3.empl \wedge y.sm > 100]\}$$

While Rel [1, 8] follows the FOI pattern for aggregation, it still inherits the pattern of using distinct aggregation scopes (i.e. separate subqueries) for each aggregate over the same relation:

```
def Q(d,av) :                                          (11)
    av = average[(e,s) : R(e,d) and S(e,s)] and
         sum[(e,s) : R(e,d) and S(e,s)] > 100
```

Figure 8 shows this relational pattern in ARC's higraph modality, and the corresponding comprehension syntax is:

$$\{Q(dept, av) \mid \quad (12)$$
$$x \in \{X(dept, av) \mid \exists r_1 \in R, s_1 \in S, \gamma_{r_1.dept}$$
$$[X.dept = r_1.dept \wedge r_1.empl = s_1.empl \wedge X.av = \mathrm{avg}(s_1.sal)]\},$$
$$y \in \{Y(dept, sm) \mid \exists r_2 \in R, s_2 \in S, \gamma_{r_2.dept}$$
$$[Y.dept = r_2.dept \wedge r_2.empl = s_2.empl \wedge Y.sm = \mathrm{sum}(s_2.sal)]\}$$
$$[Q.dept = x.dept \wedge Q.av = x.av \wedge x.dept = y.dept \wedge y.sm > 100]\}$$

Notice the similarities and differences between the relational patterns of Fig. 8/(12) and Fig. 7/(10).

[7] We made an adjustment to the query after confirming with the authors that our interpretation was correct. The aggregation as originally written in [37], $\mathrm{Aggr}_{\sum} z.(\exists y.R(x,y) \wedge S(x,z),z)$ would, on a database containing two employees with the same salary in the same department, count that salary only once rather than twice. Also, "c_{100}" is a language-specific syntax for referring to the constant 100.

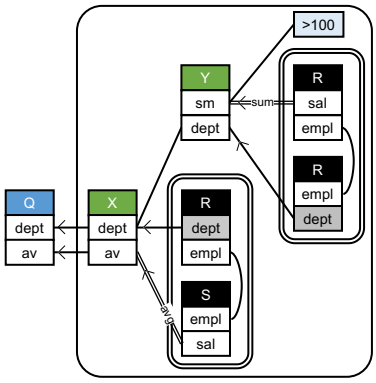


Figure 8: Pattern-preserving ARC higraph representation of the multiple-aggregate query written in Rel (11).

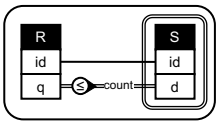
Logical sentences and integrity constraints. Expressions that evaluate to true or false can also contain aggregates. Furthermore, aggregation predicates may be comparison predicates, not assignment predicates. Figures 9b and 9d show two ARC sentences that illustrate this pattern (see [28] on how to read the negation scope):

$$\exists r \in R[\exists s \in S, \gamma_\emptyset[r.id = s.id \wedge r.q \leq \mathrm{count}(s.d)]] \quad (13)$$
$$\neg\exists r \in R[\exists s \in S, \gamma_\emptyset[r.id = s.id \wedge r.q > \mathrm{count}(s.d)]] \quad (14)$$

By contrast, the closest SQL formulations, shown in Figures 9a and 9c, can only return a unary relation representing the truth value, not a Boolean sentence directly [3, Database 737].

```
select exists(
  select 1
  from R
  where R.q <=
    (select count(S.d)
    from S
    where S.id=R.id))
```

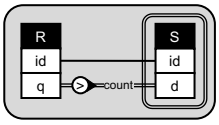
(a)



(b)

```
select not exists(
  select 1
  from R
  where R.q >
    (select count(S.d)
    from S
    where S.id=R.id))
```

(c)



(d)

Figure 9: Boolean queries and constraints (13), (14).

2.6 Language Conventions

Consider an instance with $R = \{(1,2)\}$ and $S = \emptyset$. The following Soufflé rule computes, for each $R(ak, \_)$, the sum of all b such that $S(a,b)$ and $a < ak$:

```
Q(ak,sm) :- R(ak,_), sm = sum b: {S(a,b), a<ak}.    (15)
```

On this instance, the rule derives $Q(1,0)$ because Soufflé evaluates a sum over an empty set as 0 (Soufflé has no NULL). In contrast, the equivalent SQL queries in Figs. 13a and 13b (if we add DISTINCT to the select clause) return the row $(1, \mathtt{NULL})$ on the same instance, since in SQL the result of SUM over zero input rows is NULL.

We treat such choices (how aggregates behave on empty inputs, and more generally how missing values are represented) as *conventions*. They are orthogonal to the relational structure of the query (see Fig. 13d): changing the convention affects the observable result, but not the underlying relational pattern. Accordingly, ARC abstracts from these conventions and focuses on the relational composition of queries.

2.7 Sets or bags? Not an issue for ARC but a matter of convention

Nothing needs to change *in the surface syntax of ARC* if relations are interpreted as bags (multisets) rather than sets. The conceptual evaluation still ranges over tuples as before: each tuple in one relation can be paired with each tuple in the other relation, regardless of whether a tuple has a duplicate or not. Consequently, a relational QL does not need to be designed for sets or bags; instead the same query can be *interpreted* under either set or bag semantics. Choosing set or bag interpretation is orthogonal to language design.

A common convention in the collection-types literature is to signal bag semantics by writing bag brackets (here: $\{\!\{\cdot\}\!\}$) instead of set brackets $\{\cdot\}$, e.g., $\{\!\{Q(A) \mid r \in R[Q.A = r.A]\}\!\}$ instead of $\{Q(A) \mid r \in R[Q.A = r.A]\}$. However, we treat this as a convention rather than part of the concrete syntax of query strings. The syntax of ARC does not commit to sets or bags, and the choice of semantics is fixed independently of the relational patterns expressed by the query.

Whether a query is interpreted under set or bag semantics matters for evaluation and optimization, because some rewrite rules only apply under set semantics. For example, consider the nested query

$$\{Q(A) \mid \exists r \in R[\exists s \in S[Q.A = r.A \wedge r.B = s.B]]\}$$

Under set semantics this can be unnested to

$$\{Q(A) \mid \exists r \in R, s \in S[Q.A = r.A \wedge r.B = s.B]\}$$

Under bag semantics, however, the two can differ: the nested formulation produces $Q(A)$ once per matching occurrence of r (a semijoin-like behavior), whereas the unnested formulation produces $Q(A)$ once per matching pair (r, s), which multiplies output multiplicities when multiple tuples of S share the same B-value.

Deduplication. Removing duplicates (as in DISTINCT) is expressible via grouping on all projected attributes and does not require a dedicated operator. For example, deduplicating a binary relation $R(A, B)$ can be written as:

$$\{Q(A, B) \mid \exists r \in R,\ \gamma_{r.A, r.B}[\, Q.A = r.A \wedge Q.B = r.B\,]\}.$$

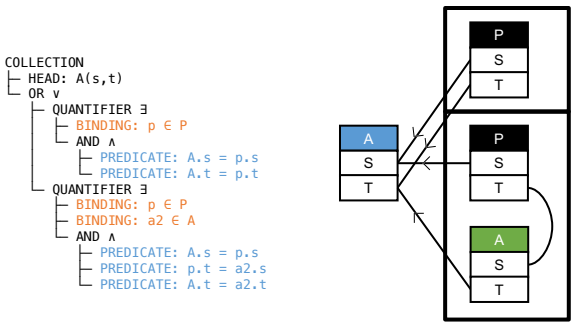
Recall that an aggregate predicate entails a grouping clause, but grouping can also appear without having an aggregate predicate.

2.8 Negation, Disjunction, and Union

Union of relations is treated as disjunction in TRC and ARC. Negation and disjunction are discussed in detail in [28].

2.9 Recursion

ARC supports recursion with the same least-fixed-point semantics as Datalog, but expressed in our *named* perspective. Let $P(s, t)$ be the parent relation, where s is the source (parent) and t is the target



(a) (b)

Figure 10: ARC representations for recursive query (16).

(child). In Datalog, the ancestor relation $A(s, t)$ is defined by the familiar two-rule program:

```
A(x,y) :- P(x,y)
A(x,y) :- P(x,z), A(z,y)
```

In Datalog, multiple rules with the same head are combined by union, and recursion is obtained by taking the least fixed point of that union. In ARC, a relation is defined by a single construct and the implicit union of multiple rules is written as a disjunction within one definition.

$$\{A(s, t) \mid \exists p \in P[A.s = p.s \wedge A.t = p.t] \vee \exists p \in P, a_2 \in A[A.s = p.s \wedge p.t = a_2.s \wedge a_2.t = A.t]\} \quad (16)$$

2.10 Null values and (NOT) IN predicates

SQL evaluates predicates in three-valued logic, so comparisons involving null may yield unknown. This interacts poorly with certain predicates, notably NOT IN. For example, the SQL query in Fig. 11a returns the empty set whenever S contains any row with null in column A, because the membership test becomes unknown and the WHERE clause filters out the row.

But this behavior can be reproduced within two-valued logic by rewriting NOT IN into NOT EXISTS while making null checks explicit [43] as in Fig. 11. We can thus replicate SQL's NULL behavior in our collection framework as well:

$$\{Q(A) \mid \exists r \in R[Q.A = r.A \wedge \neg(\exists s \in S[s.A = r.A \vee s.A \text{ is null} \vee r.A \text{ is null}])]\} \quad (17)$$

```sql
select R.A
from R
where R.A not in
  (select S.A
  from S)
```

(a) SQL: NOT IN

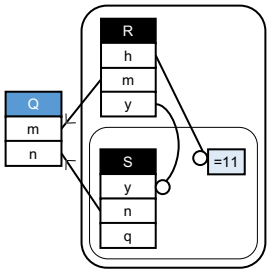
```sql
select R.A from R
where not exists
  (select 1
  from S
  where S.A=R.A
  or S.A is null
  or R.A is null)
```

(b) SQL: NOT IN with NOT EXISTS

Figure 11: Replicating SQL's null behavior for the NOT IN clause (a) with NOT EXISTS (b). (17) shows their ARC representation.

```
select R.m, S.n
from R
left outer join S
on (R.h=11 and R.y=S.y)
```

(a) Complicated outer join condition expressed in SQL



(b)

Figure 12: Outer joins and their higraph representation.

2.11 Left and full outer joins

A priori, outer joins are not naturally expressible with plain comprehensions: comprehensions range over existing collections, so a binding with no match simply disappears. For example, a left join between R and S can be written as the union of the matching and the non-matching cases:

$$\{Q(A,B) \mid \exists r \in R, s \in S[Q.A{=}r.A \wedge Q.B{=}s.B \wedge r.A{=}s.B]\} \cup$$
$$\{Q(A,B) \mid \exists r \in R[Q.A{=}r.A \wedge Q.B{=}\text{null} \wedge \neg(\exists s \in S[r.A{=}s.B])]\}$$

We therefore extend comprehensions with an explicit join annotation in the binding list (similar in spirit to the grouping operator). A join annotation specifies (*i*) which bound tables are combined by inner/left/full joins and (*ii*) the precedence (nesting) of these joins. With this extension, the left join above can be expressed as the single comprehension

$$\{Q(A,B) \mid \exists r \in R, s \in S, \text{left}(r,s)$$
$$[Q.A{=}r.A \wedge Q.B{=}s.B \wedge r.A{=}s.B]\}$$

The annotation inner is k-ary, while left and full are binary. Any scope without an explicit outer-join annotation is inner by default. For example, $\exists r \in R, s \in S, t \in T[\ldots]$ is shorthand for $\exists r \in R, s \in S, t \in T, \text{inner}(r,s,t)[\ldots]$, and an inner join followed by a left join can be written as $\exists r \in R, s \in S, t \in T, \text{left}(r, \text{inner}(s,t))[\ldots]$. Our join annotations can model arbitrary nestings of outer joins, including cases that are awkward to express in surface SQL syntax [12, 16].

At the higraph level, we depict outer join conditions by marking the optional side with an empty circle (inspired by ERD notation). Precedence scopes mirror the nesting of join annotations and can also cover cross joins. For example, Fig. 12a (from [12, example N']) corresponds to:

$$\{Q(m,n) \mid \exists r \in R, s \in S, \text{left}(r, \text{inner}(11,s)) \quad (18)$$
$$[Q.m = r.m \wedge Q.n = s.n \wedge r.y = s.y \wedge r.h = 11]\}$$

Here a literal c used as a leaf inside a join annotation denotes a singleton relation (a virtual unary table) containing just the value c; hence $\text{inner}(11,s)$ is a cross join between S and this singleton. Because outer cross joins contribute no join-condition edge in the higraph, we annotate them textually (e.g., with "×") when needed.

2.12 Representing head aggregates

Recall from Section 2.3 that ARC does not allow nesting (subqueries) in the head, and from Section 2.5 that ARC represents head aggregates as a form of lateral join in the body. We call a head aggregate

```
select R.A,
  (select sum(S.B) sm
  from S
  where S.A<R.A)
from R
```

(a) Scalar query

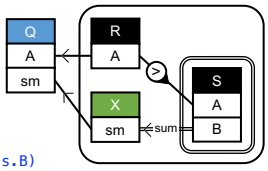
```
select R.A, X.sm
from R join lateral
  (select sum(S.B) sm
  from S
  where S.A<R.A) X
on true
```

(b) Lateral query

```
select R.A, sum(S.B) sm
from R
left join S
on S.A<R.A
group by R.A
```

(c) Left join (incorrect translation; shown as counterexample)

```
COLLECTION
├ HEAD: Q(A,sm)
└ QUANTIFIER ∃
   ├ BINDING: r ∈ R
   ├ BINDING: x ∈
   │  └ COLLECTION
   │     ├ HEAD: X(sm)
   │     └ QUANTIFIER ∃
   │        ├ BINDING: s ∈ S
   │        ├ GROUPING: ∅
   │        └ AND ∧
   │           ├ PREDICATE: s.A < r.A
   │           └ PREDICATE: X.sm = sum(s.B)
   └ AND ∧
      ├ PREDICATE: Q.A = r.A
      └ PREDICATE: Q.sm = x.sm
```



(d) Relational Pattern of a single-valued head-nested query

Figure 13: (a) A single-valued correlated scalar SQL query with an aggregate in the head. (b) An equivalent formulation that pushes the aggregate into the body using a lateral join. (c) An alternative formulation using a left outer join and GROUP BY. Only the lateral-join formulation (b) is guaranteed to preserve the semantics under both set and bag semantics (in particular, when R contains duplicates). (d): ARC does not allow nesting in the head and therefore represents such scalar queries directly in the lateral-join form (b).

in any relational language *single-valued* if, for every result tuple of the query body, the aggregate evaluates to a single scalar value (or null). This class includes SQL scalar subqueries such as Fig. 5a, as well as Soufflé head aggregates such as query (6). For those queries, the overall result is a flat relation, i.e., it contains no nested collections. Any single-valued head aggregate can be rewritten as a lateral join in the body.[8] The intuition is that a lateral join faithfully preserves the intended per-tuple semantics of a correlated scalar subquery: the inner query is re-evaluated once per outer tuple, without accidental grouping or merging. In contrast, a rewrite based on LEFT JOIN + GROUP BY [21, 23] fails to preserve the correlation pattern under bag semantics when grouping coalesces duplicates in the outer relation into a single output row.

For example, consider the single-valued SQL scalar subquery in Fig. 13a and two rewrites: as lateral join in Fig. 13b and as left join in Fig. 13c [3, Database 720]. Both rewrites are correct if the inputs contain no duplicates. Under bag semantics, however, if relation R contains duplicate values (and rows don't have a unique key), then the query in Fig. 13c collapses all identical $R.A$ values into a single group and no longer reflects the "once per tuple of R" evaluation of the subquery.[9] In contrast, the lateral join in Fig. 13b remains equivalent even under bag semantics, because the lateral

[8] This has already been observed in [17, Sect. 10] for set semantics.
[9] If each outer tuple had a unique identifier, then we could add it to the GROUP BY clause and preserve the semantics.

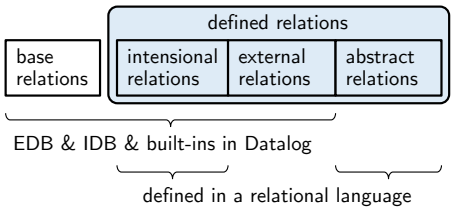


Figure 14: *Base relations* are given extensionally by enumeration. *Defined relations* are given intensionally by definitions. *Intensional relations* (views, CTEs, IDBs) are defined by relational queries and may be materialized. *External relations* (built-ins) are defined outside the relational language and may have infinite extension. *Abstract relations* are possibly domain-dependent relational expressions that help abstract and modularize large queries.

join preserves the per-outer-tuple semantics. In other words, a single-valued scalar subquery and its lateral-join encoding use the same conceptual evaluation strategy. For that reason, ARC represents scalar queries as lateral joins (Fig. 13d).

2.13 Defined relations (incl. abstract relations)

In principle, relational query languages can treat functions and arithmetic predicates uniformly as relations. Unlike *base relations* (base tables), *defined relations* are not specified extensionally by enumerating their tuples, but intensionally via a definition. Among defined relations, *intensional relations* (e.g., views and CTEs = Common Table Expressions) are definable in the relational language and, over a finite database instance, have a finite extension (and thus can be materialized). In Datalog terminology, base and intensional relations correspond to extensional and intensional predicates (EDB and IDB), respectively (Fig. 14).

2.13.1 External relations. In contrast to intensional relations, external relations (often referred to as external predicates) are defined outside the relational language and may have infinite extension. Intuitively, they correspond to built-in predicates (or built-ins) in Datalog, i.e. predefined relations that extend pure logical atoms with computational or domain-specific functionality, e.g. the arithmetic predicate "+", equality "=", comparisons such as ">", or string comparison such as SQL's "LIKE" operator.[10]

EXAMPLE 1 (ARITHMETIC AND COMPARISON OPERATORS). *A relational interpretation of the arithmetic operator "Minus" (for "−" as in 5 − 3 = 2) is given by:*

$$\{Minus(left, right, out) \mid Minus.out = Minus.left - Minus.right\}$$

Thus, in the following query with an arithmetic minus operator

$$\{Q(A) \mid \exists r \in R, s \in S, t \in T[Q.A = r.A \wedge r.B - s.B > t.B]\} \quad (19)$$

we can relationalize the minus operator (i.e., reify it as a relation) and rewrite the query. This yields a join query:

$$\{Q(A) \mid \exists r \in R, s \in S, t \in T, f \in Minus[Q.A = r.A \wedge f.left = r.B \wedge f.right = s.B \wedge f.out > t.B]\} \quad (20)$$

[10]While the comparison operator ">" may be part of a relational vocabulary, we cannot define a binary relation Bigger(A, B) containing pairs of integers where $A > B$ with relational operations alone.

```
select R.A
from R,S,T
where R.B-S.B>T.B
```

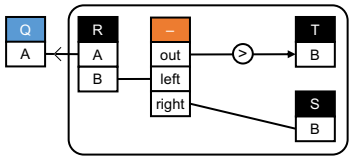
(a)

```
select R.A
from R,S,T,">","-"
where R.B="-".left
and S.B="-".right
and ">".left="-".out
and ">".right=T.B
```

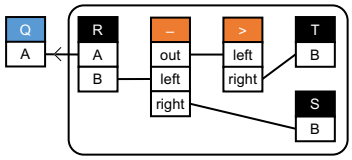
(b) named perspective

```
select R.A
from R,S,T
where "-"(R.B,S.B,x)
and ">"(x,T.B)
```

(c) positional perspective



(d) diagram for (20)



(e) diagram for Fig. 15b

Figure 15: In relational languages, every computable relation can be relationalized as an external relation with externally defined semantics. For example, (b)-(e) use an external relation for minus called "–". (b)/(c): Compare the syntax if external relations are interpreted either under a named or unnamed (positional) perspective in SQL.

We can also relationalize the comparison operator > as a separate relation named "Bigger":

$$\{Bigger(left, right) \mid Bigger.left > Bigger.right\}$$

and rewrite the query as an equijoin between relations [54]:

$$\{Q(A) \mid \exists r \in R, s \in S, t \in T, f \in Minus, g \in Bigger [Q.A = r.A \wedge f.left = r.B \wedge f.right = s.B \wedge f.out = g.left \wedge g.right = t.B]\} \quad (21)$$

Queries (19) and (21) correspond to the SQL queries from Fig. 15a and Fig. 15b, respectively. Figures 15d and 15e show the queries from (20) and (21), respectively (with minus shown as "–", etc.).

Discussion. ❶ In Example 1, the relational definition of the Minus relation uses the arithmetic operator "−". Its meaning is therefore not determined by pure relational operators and must be provided by primitives outside the relational core. Such primitives are often called "built-ins"; we use the term *external relation* to emphasize that their semantics is derived from concepts outside core relational constructs. For example, $[\![Minus]\!] = \{(x, y, z) \mid z = x - y\}$. If Add is already defined as a primitive operator, then subtraction can also be characterized via addition: $[\![Minus]\!] = \{(x, y, z) \mid Add(y, z, x)\}$. More generally, we can relationalize (reify) such operations (i.e., treat them as relations) to make their use explicit in queries. We also note that Fig. 15c illustrates a mixing of the named perspective (SQL) with an unnamed perspective where the operands of predicates are accessed positionally; such mixing can break compositionality (here, the join attribute x is not defined). The formalities of such external specifications are not our focus; we instead focus on modular building blocks of relational languages, of which external relations are one.

❷ The comprehension-style definition of *Minus* in Example 1 also raises the usual safety issue: none of its "attributes" are range-restricted, so the relation is unsafe and ill-defined. We can restore safety by guarding the operands and result with a domain relation

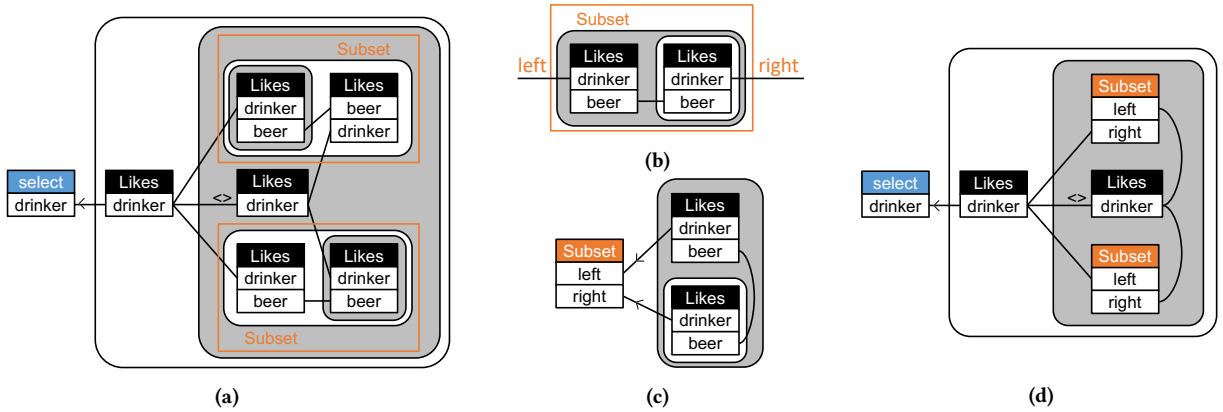


Figure 16: By using abstract relations, the unique-set query Fig. 17 can be modularized and now more easily interpreted as finding drinkers s.t. there is no other drinker who likes both a subset and a superset of the beers. Notice that the newly defined relation "Subset" does not have a well-defined extension outside the context in which it is used, and that is OK.

$D(v)$:

$$\{Minus(left, right, out) \mid \exists d_1 \in D, d_2 \in D, d_3 \in D[Minus.left = d_1.v \wedge Minus.right = d_2.v \wedge Minus.out = d_3.v \wedge d_3.v = d_1.v - d_2.v]\}$$

For our purpose, we abstract from such concrete definitions: we assume that external relations can be defined meaningfully and are then accessible to the language. Their concrete realizations are language-specific and not our focus. Under this abstraction, any computation (including arithmetic operators) can be seen as a relation, and an abstract relational query language treats computation uniformly as relations.

❸ Closely related recent work [35] formalizes external predicates (which evaluate to either true or false when all operands are fixed) as possibly infinite relations whose extensions are not stored in the database, but are accessed through specific *access patterns*.[11] Intuitively, a predicate's truth value is a function of its inputs (e.g., $Add(2, 3, 5)$ is true). Access patterns turn such Boolean predicates into a family of (multi-valued) functions that a query engine can call when only a subset of the inputs are fixed (e.g., $Add(2, x, 5)$ represents $5 - 2$ and returns $x = 3$), while still fulfilling safety requirements. This lets operands of external predicates be joined (e.g., the join "-".out = ">".left between two external predicates in Fig. 15e prevents them from being evaluated as independent Boolean predicates) and also enables such predicates to produce outputs when connected via assignment predicates (see e.g., Section 3.1 and Fig. 20). In other words, richer safety conditions allow these predicates to be treated like ordinary database relations during query evaluation.

❹ Other recent work [28] relationalizes join and selection predicates into "anchor relations" in order to support arbitrarily nested disjunctions in a diagrammatic presentation (higraph modality).

[11] Notice a slightly different motivation for the word "external predicate": For [35], external predicates are "computed on demand rather than stored", they are *external* to the database, and only usable through a controlled interface (access patterns). We use the word external to focus on the fact that their formal definition needs to bring in concepts that are external to the relational model and cannot be described by standard relational operators. Both interpretations agree that operations can be relationalized (reified, i.e. turned into relations) for the purpose of analyzing and describing queries.

2.13.2 Abstract Relations. Abstract relations are relation symbols defined *within* a relational language to name and abstract a subquery. In contrast to *external relations*, abstract relations need not denote a standalone, well-defined extension on their own. In particular, an abstract relation may be domain-dependent and thus may not have a well-defined extension on its own. Nevertheless, when an abstract relation occurs inside a safe surrounding query, it can be interpreted as denoting *some* reasonable finite relation that makes the overall query well-defined. This is exactly the point of abstraction: when analyzing the intent of the larger query, we do not need to reason about the internal details of the module or its standalone extension.

EXAMPLE 2 (UNIQUE-SET QUERY). *We are given a single relation Likes(drinker, beer), which we abbreviate by $L(d, b)$, and wish to find drinkers who like a unique set of beers, i.e., no other drinker likes the exact same set of beers (see [41, Fig. 1], [31, Fig. 9] for extensive discussion of this query). (22) In the relationally complete fragment (the first-order fragment), the query is written as Fig. 17 in SQL and as follows in TRC and thus also in ARC:*

$$\{Q(d) \mid \exists l_1 \in L[Q.d = l_1.d \wedge \neg(\exists l_2 \in L[l_2.d <> l_1.d \wedge \neg(\exists l_3 \in L[l_3.d = l_2.d \wedge \neg(\exists l_4 \in L[l_4.b = l_3.b \wedge l_4.d = l_1.d])] \wedge \neg(\exists l_5 \in L[l_5.d = l_1.d \wedge \neg(\exists l_6 \in L[l_6.d = l_2.d \wedge l_6.b = l_5.b])])])]\} \quad (22)$$

To modularize the query, we define an abstract relation Subset (denoted S in ARC):

$$\{S(left, right) \mid \neg(\exists l_3 \in L[l_3.d = S.left \wedge \neg(\exists l_4 \in L[l_4.b = l_3.b \wedge l_4.d = S.right])])\} \quad (23)$$

Taken in isolation, that definition is not safe and therefore does not define a view with a well-defined extension. But in the context of the enclosing query, the module represents exactly the intended subset relation between drinkers and allows us to modularize and

```
select distinct L1.drinker
from Likes L1
where not exists
  (select 1
  from Likes L2
  where L1.drinker <> L2.drinker
  and not exists
    (select 1
    from Likes L3
    where L3.drinker = L2.drinker
    and not exists
      (select 1
      from Likes L4
      where L4.drinker = L1.drinker
      and L4.beer = L3.beer))
  and not exists
    (select 1
    from Likes L5
    where L5. drinker = L1.drinker
    and not exists
      (select 1
      from Likes L6
      where L6.drinker = L2.drinker
      and L6.beer = L5.beer)))
```

Figure 17: Unique-set query (Example 2).

compartmentalize the query. Abstracting it as such we can use it and rewrite the original query more concisely as:

$$\{Q(d) \mid \exists l_1 \in L[Q.d = l_1.d \wedge \neg(\exists l_2 \in L, s_1 \in S, s_2 \in S[l_2.d <> l_1.d \wedge s_1.left = l_1.d \wedge s_1.right = l_2.d \wedge s_2.left = l_2.d \wedge s_2.right = l_1.d])]\} \quad (24)$$

In the diagrammatic modality, abstract relations correspond to sub-diagrams that can be *collapsed* and *expanded*: a complex sub-structure (including its internal scopes) can be replaced by a clearly distinguished module node labeled with the abstract relation name, and later expanded again. This supports complexity management for large queries via modularization, hierarchy, and "zooming" (see also [45, Sect. 4.4]).

3 TWO EXAMPLES

3.1 Matrix multiplication

We now illustrate the matrix-multiplication example from the Rel paper [8] Rel expresses matrix multiplication between matrices A and B in sparse relational form and domain-based positional notation as follows:

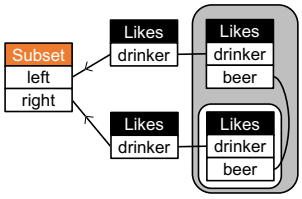
```
def MatrixMult[i,j] :                                    (25)
    sum[[k] : A[i,k]*B[k,j]]
```

If we allow arithmetic operations in ARC and assume all matrices use the same schema (row, col, val), the same computation can be written in the named perspective as:

$$\{C(row, col, val) \mid \exists a \in A, b \in B, \gamma_{a.row,b.col} [C.row = a.row \wedge C.col = b.col \wedge a.col = b.row \wedge C.val = \text{sum}(a.val * b.val)]\}$$

```
select distinct D1.drinker as left,
                D2.drinker as right
into Subset
from Likes D1, Likes D2
where not exists
  (select 1
  from Likes L3
  where not exists
    (select 1
    from Likes L4
    where L4.beer = L3.beer
    and D2.drinker = L4.drinker)
  and D1.drinker = L3.drinker)
```

(a)



(b)

Figure 18: Safely defined Subset relation for (Example 2).

```
select distinct L1.drinker
from Likes L1
where not exists
  (select 1
  from Likes L2, Subset S1, Subset S2
  where L1.drinker <> L2.drinker
  and S1.left=L1.drinker
  and S1.right=L2.drinker
  and S2.left=L2.drinker
  and S2.right=L1.drinker)
```

Figure 19: Query from (Example 2) rewritten to use the view from Fig. 18.

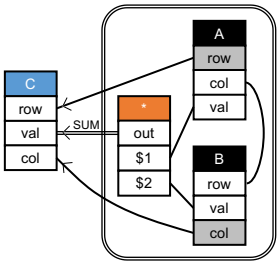


Figure 20: Matrix multiplication (25)/(26) in the higraph modality.

In the higraph modality Fig. 20, multiplication is modeled via an external relation "*"($1, $2, out):

$$\{C(row, col, val) \mid \exists a \in A, b \in B, f \in "*", \gamma_{a.row,b.col} [C.row = a.row \wedge C.col = b.col \wedge a.col = b.row \wedge C.val = \text{sum}(f.out) \wedge f.\$1 = a.val \wedge f.\$2 = b.val]\} \quad (26)$$

3.2 An illustration of the count bug

The count bug [24] is a famous example of an attempted reformulation of a nested correlated query like the one in Fig. 21a and replacing it with Fig. 21b (the incorrect translation was given in

```
select R.id
from R
where R.q =
  (select count(S.d)
  from S
  where S.id = R.id)
```

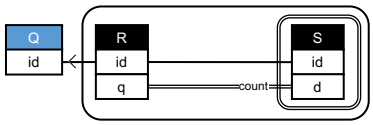
(a) Count bug: version 1

```
select R.id
from R,
  (select S.id, count(S.d) as ct
  from S
  group by S.id) as X
where R.q = X.ct and R.id = X.id
```

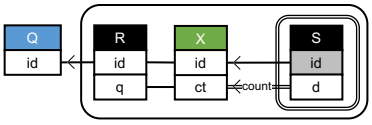
(b) Count bug: version 2

```
select R.id
from R,
  (select R2.id, count(S.d) as ct
  from R R2 left join S
  on R2.id = S.id
  group by R2.id) as X
where R.q = X.ct and R.id = X.id
```

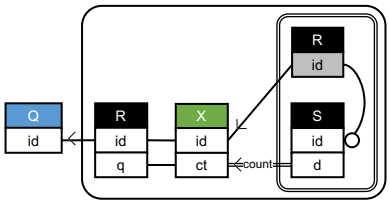
(c) Count bug: version 3



(d) Count bug: version 1



(e) Count bug: version 2



(f) Count bug: version 3

```
COLLECTION
├─ HEAD: Q(id)
└─ QUANTIFIER ∃
   ├─ BINDING: r ∈ R
   └─ AND ∧
      ├─ PREDICATE: Q.id = r.id
      └─ QUANTIFIER ∃
         ├─ BINDING: s ∈ S
         ├─ GROUPING: ∅
         └─ AND ∧
            ├─ PREDICATE: r.id = s.id
            └─ PREDICATE: r.q = count(s.d)
```

(g) ALT version 1

```
COLLECTION
├─ HEAD: Q(id)
└─ QUANTIFIER ∃
   ├─ BINDING: r ∈ R
   ├─ BINDING: x ∈
   └─ COLLECTION
      ├─ HEAD: X(id, ct)
      └─ QUANTIFIER ∃
         ├─ BINDING: s ∈ S
         ├─ GROUPING: s.id
         └─ AND ∧
            ├─ PREDICATE: X.id = s.id
            └─ PREDICATE: X.ct = count(s.d)
   └─ AND ∧
      ├─ PREDICATE: Q.id = r.id
      ├─ PREDICATE: r.id = x.id
      └─ PREDICATE: r.q = x.ct
```

(h) ALT version 2

```
COLLECTION
├─ HEAD: Q(id)
└─ QUANTIFIER ∃
   ├─ BINDING: r ∈ R
   ├─ BINDING: x ∈
   └─ COLLECTION
      ├─ HEAD: X(id, ct)
      └─ QUANTIFIER ∃
         ├─ BINDING: r2 ∈ R
         ├─ BINDING: s ∈ S
         ├─ GROUPING: r2.id
         ├─ JOIN: left(r2, s)
         └─ AND ∧
            ├─ PREDICATE: X.id = r2.id
            ├─ PREDICATE: X.ct = count(s.d)
            └─ PREDICATE: r2.id = s.id
   └─ AND ∧
      ├─ PREDICATE: Q.id = r.id
      ├─ PREDICATE: r.id = x.id
      └─ PREDICATE: r.q = x.ct
```

(i) ALT version 3

Figure 21: Section 3.2: Illustrations of the count bug: Left/middle/right columns correspond to queries (27)/(28)/(29), respectively.

[39], and corrected in [24]) However, on an input database with $R(9, 0)$ and empty table S, the first query would return 9, whereas the second would return an empty result. The correct decorrelation happens with left join and the query shown in Fig. 21c.[12] The remaining equations and figures in this section show those queries in a pattern-equivalent ARC representations and various modalities.

$$\{Q(id) \mid \exists r \in R[Q.id = r.id \wedge \exists s \in S, \gamma_{\emptyset} \quad (27)$$
$$[r.id = s.id \wedge r.q = \text{count}(s.d)]]\}$$

$$\{Q(id) \mid \exists r \in R, x \in \{X(id, ct) \mid \quad (28)$$
$$\exists s \in S, \gamma_{s.id}[X.id = s.id \wedge X.ct = \text{count}(s.d)]\}$$
$$[Q.id = r.id \wedge r.id = x.id \wedge r.q = x.ct]\}$$

$$\{Q(id) \mid \exists r \in R, x \in \{X(id, ct) \mid \exists s \in S, r_2 \in R, \gamma_{r_2.id}, \text{left}(r_2, s) \quad (29)$$
$$[X.id = r_2.id \wedge X.ct = \text{count}(s.d) \wedge r_2.id = s.id]\}$$
$$[Q.id = r.id \wedge r.id = x.id \wedge r.q = x.ct]\}$$

4 QUESTIONS & ANSWERS

The flat relational model is obsolete. We need to think bigger *and move to an Entity-Relational Data model ERDs [18], relational maps [19], databases as output [47], or at least a nested relational model [10].* While we see merit in these directions, our goal is more modest. Rather than replace or extend the relational model, our goal is to complement current practices. SQL remains widely used, and when inputs and outputs are in 1NF, nesting of intermediate results adds no expressive power [42]. That said, nested relations are now part of standards [10] and the development of an abstract nested relational QL remains open.

SQL is bad for users. We need to create a new language*, like SaneQL [46], Rel [8], or a pipe/dataflow syntax [52].* That may or may not be true, but our goal is orthogonal: rather than propose yet another QL or extension to SQL, we suggest that the database community instead creates a relational *reference language* (a relational *meta language*) that abstracts the relational patterns across query languages (the relational core) away from a surface syntax.

Usability is not an immutable property of a language, it also depends on the *modality* in which it is presented. Therefore, ARC provides multiple modalities, including a machine-oriented data structure called *Abstract Language Tree* (ALT). Because ALT exposes bindings, scopes, and grouping structure directly, it supports systematic traversal, rewriting, and validation, and can serve as an intermediate representation for NL2SQL systems that translate natural language into query intent and then render to SQL. Ultimately, questions about relative usability need to be solved with reproducible, task-oriented user studies [29, 55].

Is this then another Intermediate Representation? Our goal is fundamentally different from intermediate representations (IRs) like Semiring Dictionaries [50], Substrait [5], and relational maps [19],

[12] The example assumes R.id is a key. Otherwise, the correct translation requires an additional deduplication.

which are typically closely tied to execution models, and designed to support optimization. In contrast, we aim to move in the opposite direction, toward a more abstract representation that decouples from both data layout and syntax, focusing instead on the semantic structure of relational queries.

Is this about logical expressiveness with aggregates? While the addition of aggregate functions to logic has traditionally been studied in terms of logical expressiveness [42], our goal is different. Our focus is to capture relational patterns across various languages. For that purpose we designed a reference language that supports aggregate queries with the same relational patterns as SQL, including multiple aggregates evaluated within a single grouping scope.

Are you reinventing Rel? No. We fully embrace the Rel philosophy [1, 8]: everything is a relation, and some relations are defined (or derived) rather than stored. Rel aims to unify data modeling, querying, and application logic within a single relational language, removing the boundary between a relational query sublanguage (e.g., SQL) and the host application language (e.g., Java) so that a powerful execution engine can optimize globally across the entire program. Our aim is not to design such a language or an execution engine. Instead, our goal is to support the ongoing discussion about both user-facing and machine-facing language design trade-offs, the different patterns that appear between relational languages, and the recurring relational core expressible in all of them.

To this end, we provide an expressive *abstract relational query language* that comes with a pattern-preserving diagrammatic "modality". This language is intended as a *reference language* for analyzing current and future relational languages, whether they adopt set or bag semantics (which we regard as a "convention" orthogonal to a language itself).

Why are we still talking about new languages if everything will be NL2SQL anyway? ARC/ALT can be used as an intermediate target: models generate a structurally constrained representation, which can be validated (well-scoped variables, grouping legality, correlation shape) and then rendered to SQL. This enables intent-based evaluation and comparison of generated queries at the semantic-structure level rather than at the surface-syntax level.

Do you really expect people to write queries in a complicated looking formalism like (29)? No. Our point is not to replace SQL, Datalog, or any other surface syntax. The point is to make query meaning sayable. We propose *a universal relational reference language*, paired with a *shared vocabulary*, that names the primitive operations by which queries combine relations to answer relational questions. Once those primitives are named, we can talk about the same underlying patterns across very different languages (declarative, procedural, or functional) without mistaking syntax for substance. Thus, a reference language must be explicit, and explicitness is often verbose: what production languages compress into syntactic sugar, convention, or "obvious" readings must be surfaced if we want a reliable reference point for comparison. That surfacing is not busywork; it allows us to name distinctions that our current vocabulary blurs. It lets us point at a query in Soufflé and say "FOI aggregation." It lets us look at Fig. 20 and see the relational pattern for matrix multiplication. And it lets us diagnose bugs by naming the difference between an aggregate used as a value (assignment predicate) and an aggregate used as a test (comparison predicate), as in the count bug (Fig. 21).

And anyway, fewer people will write queries directly in the future; more people will read them and try to make sense of them. In that setting, usability is not just the language, it is about the modality in which it is presented. The same semantics can be rendered as text, diagrams, or ALTs; different modalities serve different readers. For humans, initial evidence suggests that a diagrammatic modality like Fig. 21f can be read faster and more accurately than SQL [29, 55]. For machines (including LLMs), we believe that explicit, modular structure with small, reusable vocabulary can improve precision and recall.

5 NEXT STEPS

We view Abstract Relational Calculus (ARC) as a candidate *semantic backbone*: a relational metalanguage for connecting the surface syntax of queries to their core relational intent, across query languages, modalities, and conventions.

On the systems side, we are building a SQL ↔ ARC translator that can render ARC in all 3 modalities, extending our prototype implementation [48] to cover additional aggregation-nesting patterns and disjunctions. A spring 2026 seminar on relational language design [6] focuses on the pattern expressiveness of relational languages and, in the process, produce further embeddings of other languages in ARC.

On the theory side, we plan to prove coverage results: for a well-defined fragment of SQL (including arbitrarily correlated queries and aggregation patterns), every query has a pattern-preserving ARC representation, and thus SQL ↔ ARC round-tripping is semantics-preserving (with appropriate conventions).

Finally, an open question is whether ARC/ALT can indeed serve as an effective intermediate target for NL2SQL, together with datasets and evaluation metrics that score intent via semantic structure proxies (scopes, joins, relational patterns) rather than SQL syntax similarity. Also open are extensions to sorted lists (ORDER BY), as well as extensions to the nested relational model [10].

6 ACKNOWLEDGEMENTS

We thank Molham Aref [8], Torsten Grust [33], Leonid Libkin [8, 37], Wim Martens [8], Amir Shaikhha [50], and Dan Suciu [9], for taking the time to discuss their respective papers with us, and Mahmoud Abo Khamis for helping us understand Rel queries. Wolfgang was supported in part by the National Science Foundation (NSF) under award IIS-1762268, and Diandre by the NSF Graduate Research Fellowship Program (GRFP).

REFERENCES

[1] 2025. Rel Cheatsheet. https://rel.relational.ai/rel/cheatsheet
[2] 2025. Soufflé. Aggregates and Generative Functors. https://souffle-lang.github.io/aggregates
[3] 2025. SQL activities for cs3200 and cs7240 and cs7575. https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql (Copy and paste the respective SQL commands into PostgreSQL to run the SQL queries shown in the paper).
[4] 2025. SQLGlot Abstract Syntax Tree (AST) Viewer. https://sqlglot-ast-viewer.streamlit.app/
[5] 2025. Substrait: Cross-Language Serialization for Relational Algebra. https://substrait.io/
[6] 2026. cs7575: A Seminar On Relational Language Design (Spring 2026). https://northeastern-datalab.github.io/cs7575/sp26/
[7] Anastasia Ailamaki, Samuel Madden, Daniel Abadi, Gustavo Alonso, Sihem Amer-Yahia, Magdalena Balazinska, Philip A. Bernstein, Peter A. Boncz, Michael J. Cafarella, Surajit Chaudhuri, Susan B. Davidson, David J. DeWitt, Yanlei Diao,

Xin Luna Dong, Michael J. Franklin, Juliana Freire, Johannes Gehrke, Alon Y. Halevy, Joseph M. Hellerstein, Mark D. Hill, Stratos Idreos, Yannis E. Ioannidis, Christoph Koch, Donald Kossmann, Tim Kraska, Arun Kumar, Guoliang Li, Volker Markl, Renée J. Miller, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Aditya G. Parameswaran, Ippokratis Pandis, Jignesh M. Patel, Andrew Pavlo, Danica Porobic, Viktor Sanca, Michael Stonebraker, Julia Stoyanovich, Dan Suciu, Wang-Chiew Tan, Shivaram Venkataraman, Matei Zaharia, and Stanley B. Zdonik. 2025. The Cambridge Report on Database Research. *CoRR* abs/2504.11259 (2025). doi:10.48550/ARXIV.2504.11259

[8] Molham Aref, Paolo Guagliardo, George Kastrinis, Leonid Libkin, Victor Marsault, Wim Martens, Mary McGrath, Filip Murlak, Nathaniel Nystrom, Liat Peterfreund, Allison Rogers, Cristina Sirangelo, Domagoj Vrgoc, David Zhao, and Abdul Zreika. 2025. Rel: A Programming Language for Relational Data. In *Companion of SIGMOD/PODS 2025*. 283–296. doi:10.1145/3722212.3724450

[9] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. 1994. Comprehension Syntax. *SIGMOD Record* 23, 1 (1994), 87–96. doi:10.1145/181550.181564

[10] Michael J. Carey, Don Chamberlin, Almann Goo, Kian Win Ong, Yannis Papakonstantinou, Chris Suver, Sitaram Vemulapalli, and Till Westmann. 2024. SQL++: We Can Finally Relax!. In *ICDE*. 5501–5510. doi:10.1109/ICDE60146.2024.00438

[11] Edgar F. Codd. 1982. Relational Database: A Practical Foundation for Productivity. *CACM* 25, 2 (1982), 109–117. doi:10.1145/358396.358400

[12] Maria Colgan. 2023. Outerjoins in Oracle. https://blogs.oracle.com/optimizer/post/outerjoins-in-oracle

[13] Keith D. Cooper and Linda Torczon. 2022. *Engineering a compiler* (3 ed.). Morgan Kaufmann. doi:10.1016/C2014-0-01395-0

[14] Jonathan Danaparamita and Wolfgang Gatterbauer. 2011. QueryViz: helping users understand SQL queries and their patterns. In *EDBT*. 558–561. doi:10.1145/1951365.1951440 Project page: https://queryvis.com/.

[15] Christopher J. Date. 2004. *An introduction to database systems* (8 ed.). Pearson/Addison Wesley Longman. https://dl.acm.org/doi/10.5555/861613

[16] Michael M David. 1999. *Advanced ANSI SQL data modeling and structure processing*. Artech House, Boston.

[17] Jan Van den Bussche and Stijn Vansummeren. 2009. Translating SQL into the relational algebra. Course notes, Hasselt University and Université Libre de Bruxelles. https://cs.ulb.ac.be/public/_media/teaching/infoh417/sql2alg_eng.pdf

[18] Amol Deshpande. 2025. Beyond Relations: A Case for Elevating to the Entity-Relationship Abstraction, In CIDR. *CoRR*. doi:10.48550/ARXIV.2505.03536

[19] Jens Dittrich. 2025. How to get Rid of SQL, Relational Algebra, the Relational Model, ERM, and ORMs in a Single Paper - A Thought Experiment. *CoRR* (2025). doi:10.48550/ARXIV.2504.12953

[20] Ramez Elmasri and Sham Navathe. 2015. *Fundamentals of database systems* (7 ed.). Addison Wesley. https://dl.acm.org/doi/book/10.5555/2842853

[21] Leonidas Fegaras and David Maier. 2000. Optimizing object queries using an effective calculus. *TODS* 25, 4 (2000), 457–516. doi:10.1145/377674.37767

[22] Avrilia Floratou, Fotis Psallidas, Fuheng Zhao, Shaleen Deep, Gunther Hagleither, Wangda Tan, Joyce Cahoon, Rana Alotaibi, Jordan Henkel, Abhik Singla, Alex Van Grootel, Brandon Chow, Kai Deng, Katherine Lin, Marcos Campos, K. Venkatesh Emani, Vivek Pandit, Victor Shnayder, Wenjing Wang, and Carlo Curino. 2024. NL2SQL is a solved problem... Not!. In *CIDR*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p74-floratou.pdf

[23] César Galindo-Legaria and Milind Joshi. 2001. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*. 571–581. doi:10.1145/375663.375748

[24] Richard A. Ganski and Harry K. T. Wong. 1987. Optimization of Nested SQL Queries Revisited. In *SIGMOD*. 23–33. doi:10.1145/38713.38723

[25] Wolfgang Gatterbauer. 2011. Databases will Visualize Queries too. *PVLDB* 4, 12 (2011), 1498–1501. doi:10.14778/3402755.3402805 Recorded Talk: https://www.youtube.com/watch?v=kVFnQRGAQls. Slides: https://gatterbauer.name/download/vldb2011_Database_Query_Visualization_presentation.pdf. Project page: https://queryvis.com/.

[26] Wolfgang Gatterbauer. 2023. A Tutorial on Visual Representations of Relational Queries. *PVLDB* 16, 12 (2023), 3890–3893. doi:10.14778/3611540.3611578 Paper: https://www.vldb.org/pvldb/vol16/p3890-gatterbauer.pdf. Tutorial page: https://northeastern-datalab.github.io/visual-query-representation-tutorial/. Slides: https://northeastern-datalab.github.io/visual-query-representation-tutorial/slides/VLDB_2023-Visual_Representations_of_Relational_Queries.pdf.

[27] Wolfgang Gatterbauer. 2024. A Comprehensive Tutorial on over 100 Years of Diagrammatic Representations of Logical Statements and Relational Queries. In *ICDE*. IEEE. doi:10.1109/ICDE60146.2024.00407 Paper: https://arxiv.org/pdf/2404.00007. Tutorial page: https://northeastern-datalab.github.io/diagrammatic-representation-tutorial/. Slides: https://northeastern-datalab.github.io/diagrammatic-representation-tutorial/ICDE_2024-Diagrammatic-Representations-Tutorial.pdf.

[28] Wolfgang Gatterbauer. 2024. A Principled Solution to the Disjunction Problem of Diagrammatic Query Representations. SIGMOD 2026 (to appear). doi:10.48550/ARXIV.2412.08583

[29] Wolfgang Gatterbauer and Cody Dunne. 2024. On The Reasonable Effectiveness of Relational Diagrams: Explaining Relational Query Patterns and the Pattern Expressiveness of Relational Languages. *PACMMOD* 2, 1, Article 61 (2024). doi:10.1145/3639316 Recorded video: https://www.youtube.com/watch?v=IVRPD-074Ro. Slides: https://gatterbauer.name/download/sigmod2024-Relational-Diagrams-slides.pdf. Project page: https://relationaldiagrams.com/. Main supplemental material folder on OSF: https://osf.io/q9g6u/. Online appendix with all proofs, further illustrations, and study materials: https://arxiv.org/pdf/2401.04758. Textbook analysis: https://osf.io/u7c4z. User study tutorial: https://osf.io/mruzw. Stimuli-generating code: https://osf.io/kgx4y. The stimuli: https://osf.io/d5qaj. Stimuli/schema index CSV: https://osf.io/u8bf9. Stimuli/schema index JSON: https://osf.io/sn83j. Server code for hosting the study: https://osf.io/suj4a. Collected data: https://osf.io/8vm42. Executed user study analysis code: https://osf.io/f2xe3. Preregistered user study: https://osf.io/4zpsk/. Results from SIGMOD 2024 ARI (Availability & Reproducibility Initiative): https://reproducibility.sigmod.org/reports.html.

[30] Wolfgang Gatterbauer and Cody Dunne. 2025. Relational Diagrams and the Pattern Expressiveness of Relational Languages. *SIGMOD Record* 54, 1 (2025), 80–88. doi:10.1145/3733620.3733637 Paper: https://sigmodrecord.org/?smd_process_download=1&download_id=14010. Project page: https://relationaldiagrams.com/.

[31] Wolfgang Gatterbauer, Cody Dunne, H. V. Jagadish, and Mirek Riedewald. 2022. Principles of Query Visualization. *IEEE Data Eng. Bull.* 45, 3 (2022), 47–67. http://sites.computer.org/debull/A22sept/p47.pdf

[32] Thomas R G Green. 1990. Cognitive dimensions of notations. In *People and Computers V*. Cambridge University Press, USA, 443–460. https://dl.acm.org/doi/10.5555/92968.93015

[33] Torsten Grust and Marc H. Scholl. 1999. How to Comprehend Queries Functionally. *J. Intell. Inf. Syst.* 12, 2-3 (1999), 191–218. doi:10.1023/A:1008705026446

[34] Torsten Grust and Amir Shaikhha (Eds.). 2025. *The 19th International Symposium on Database Programming Languages (DBPL)*. https://sites.google.com/view/dbpl2025/

[35] Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, and Cristina Sirangelo. 2025. Queries with External Predicates. In *ICDT (LIPIcs, Vol. 328)*. 22:1–22:20. doi:10.4230/LIPICS.ICDT.2025.22

[36] David Harel. 1988. On Visual Formalisms. *CACM* 31, 5 (1988), 514–530.

[37] Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. 2001. Logics with aggregate operators. *JACM* 48, 4 (2001), 880–907. doi:10.1145/502090.502100

[38] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *PODS*. 13–28. https://doi.org/10.1145/2902251.2902280

[39] Won Kim. 1982. On Optimizing an SQL-like Nested Query. *TODS* 7, 3 (1982), 443–469. doi:10.1145/319732.319745

[40] Anthony C. Klug. 1982. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *JACM* 29, 3 (1982), 699–717. doi:10.1145/322326.322332

[41] Aristotelis Leventidis, Jiahui Zhang, Cody Dunne, Wolfgang Gatterbauer, H. V. Jagadish, and Mirek Riedewald. 2020. QueryVis: Logic-based Diagrams help Users Understand Complicated SQL Queries Faster. In *SIGMOD*. 2303–2318. doi:10.1145/3318464.3389767

[42] Leonid Libkin. 2003. Expressive power of SQL. *Theor. Comput. Sci.* 296, 3 (2003), 379–404. doi:10.1016/S0304-3975(02)00736-3

[43] Leonid Libkin and Liat Peterfreund. 2023. SQL Nulls and Two-Valued Logic. In *PODS*. 11–20. doi:10.1145/3584372.3588661

[44] Guido Moerkotte. 2025. Building query compilers. https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf

[45] Daniel L. Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Trans. Software Eng.* 35, 6 (2009), 756–779. doi:10.1109/TSE.2009.67

[46] Thomas Neumann and Viktor Leis. 2024. A Critique of Modern SQL and a Proposal Towards a Simple and Expressive Query Language. In *CIDR*. https://www.cidrdb.org/cidr2024/papers/p48-neumann.pdf

[47] Joris Nix and Jens Dittrich. 2025. Extending SQL to Return a Subdatabase. *PACMMOD* 3, 3 (2025), 154:1–154:26. doi:10.1145/3725291

[48] Diandre Miguel Sabale and Wolfgang Gatterbauer. 2025. PatternVis: A Tool for Relational Pattern Visualization. In *Companion of the SIGMOD 2025*. 227–230. doi:10.1145/3722212.3725128

[49] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*. ACM, 196–206. doi:10.1145/2892208.2892226

[50] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *PACMPL (OOPSLA1)* 6 (2022), 1–33. doi:10.1145/3527333

[51] Mary Shaw and Michael Hilton. 2025. *You're a Computer Science Major. Don't Panic.* NYTimes (Nov. 11, 2025). https://www.nytimes.com/2025/11/12/opinion/ai-coding-computer-science.html (Don't just read the article, also read the top-rated user comments).

[52] Jeff Shute, Shannon Bales, Matthew Brown, Jean-Daniel Browne, Brandon Dolphin, Romit Kudtarkar, Andrey Litvinov, Jingchi Ma, John D. Morcos,

Michael Shen, David Wilhite, Xi Wu, and Lulan Yu. 2024. SQL has problems. We can fix them: Pipe syntax in SQL. *PVLDB* 17, 12 (2024), 4051–4063. doi:10.14778/3685800.3685826

[53] Philip W. Trinder. 1991. Comprehensions, a Query Notation for DBPLs. In *DBPL3*. 55–68. https://dl.acm.org/doi/10.5555/135260.135271

[54] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2021. Beyond Equi-joins: Ranking, Enumeration and Factorization. *PVLDB* 14, 11 (2021), 2599–2612. http://www.vldb.org/pvldb/vol14/p2599-tziavelis.pdf

[55] Giorgio Vinciguerra, Guang Yang, Wolfgang Gatterbauer, and Cody Dunne. 2025. Reproducibility Report for ACM SIGMOD 2024 Paper: 'On The Reasonable Effectiveness of Relational Diagrams'. In *Reproducibility Reports of SIGMOD 2024*. 60–63. doi:10.1145/3687998.3717044

[56] Iaroslav Zeigerman. 2023. Semantic Understanding of SQL. https://www.tobikodata.com/blog/semantic-understanding-of-sql Blog post (May 10, 2023).