# Relational Programming in Rel

Molham Aref[1], Paolo Guagliardo[2], George Kastrinis[1], Leonid Libkin[1,2,7], Victor Marsault[3], Wim Martens[1,4], Mary McGrath[1], Filip Murlak[5], Nathaniel Nystrom[1], Liat Peterfreund[6], Allison Rogers[1], Cristina Sirangelo[1,7], Domagoj Vrgoč[8], David Zhao[1], Abdul Zreika[1]
[1]RelationalAI [2]University of Edinburgh [3]LIGM, Univ. Gustave Eiffel, CNRS [4]University of Bayreuth [5]University of Warsaw [6]Hebrew University [7]IRIF, Université Paris Cité, CNRS [8]PUC Chile

## ABSTRACT

From the moment of their inception, languages for relational data have been described as *sublanguages* embedded in a host programming language. Rel is a new relational language that goes beyond this paradigm, with features that allow for programming in the large, making it possible to fully describe end to end application semantics. With the new approach we can model the semantics of entire enterprise applications relationally, which helps significantly reduce architecture complexity and avoid the well-known *impedance mismatch* problem. This paradigm shift is enabled by 50 years of database research, making it possible to revisit the sublanguage/host language paradigm, starting from the fundamental principles. We present a gentle introduction to Rel and the principles behind its design philosophy.

## 1. INTRODUCTION

Languages for relational data have always been described as *sublanguages* [9, 10]. This means that such languages are not designed to describe the entire end-to-end application logic of programs that involve data, but rather focus on specific operations that concern the storage, retrieval, or manipulation of data within such programs. This view of database languages made a lot of sense fifty years ago, when database query languages were being introduced in an environment of procedural imperative programming, when declarative languages were a revolutionary idea, and when it was unclear if this idea could be extended beyond its domain-specific use. Today, SQL is the prime success story of a declarative and domain-specific language. It remains in high demand by tech employers [7], which is a remarkable achievement for a 40-year-old programming (sub)language.

The sublanguage paradigm however has important limitations. To start with, it entails having to communicate with a host programming language. This causes the well-known *impedance mismatch*: the two languages have different data types, different data structures, different memory models, etc. Often they are even based on *different programming paradigms*, one being fundamentally declarative and the other fundamentally imperative. Furthermore, the two languages typically have separate runtime environments, which limits the optimizations that can be applied to programs that use both languages. The host language runtime environment may not support features of the database such as automatic out of core computation, automatic parallelism, incremental computation. In other words, the impedance mismatch:

1. causes extra complexity for developers, unrelated to solving the problem at hand;
2. limits the capabilities of automatic support that we are used to in databases.

Another effect of the sublanguage paradigm is that query languages are not equipped with important features needed for programming in the large, as these are delegated to the host language. Most notably, query languages lack support for building libraries. Indeed, SQL does not have a standard library, and when new features are needed they are added by means of expanding the language itself. Over the years, this led to an 11-part ISO standard comprising well over 4000 pages which is an order of magnitude larger than the C standard. This breaks a fundamental principle of programming language design formulated by Steele [25]: *define a small core and provide the functionality to build libraries.*

Rel aims to go beyond the sublanguage paradigm. The database industry and research communities have accumulated a vast array of insights that reshuffle the cards on which the sublanguage design decision was based. We ask ourselves, can we *design and implement* a language that natively handles data and semantics in a database, preserving the bedrock principles of databases (such as the data independence, communicability, and set-at-a-time processing objectives [12]), providing the programmer with the necessary constructs to factor semantics out of application programs? Such a full-featured language for data and semantics would allow for new powerful simplifications and optimizations owing to a single runtime environment, increasing the productivity of users and application developers.

In this paper, we give a gentle introduction to Rel while presenting our approach to language design. With it, we aim to provide language support for enterprise AI that brings semantics close to data and drastically simplifies application development. We build upon the foundational ideas behind the relational model — data modeling via normalization and declarative programming — and take them further. On the data modeling front, we advocate for *full normalization* as a means of coupling data and semantics. On the language front, we advocate for extending the declarative paradigm to the entire language, beyond its data management fragment.

Rel has been designed and implemented based on those

principles. Its key features are:

1. manipulation of logical formulas and entire relations;
2. powerful *recursion* built on the foundations of Datalog;
3. tools that enable writing libraries:
   - *abstraction* and *application* as key constructs;
   - variables that can *range over tuples and relations*.

We touch upon all these points in the course of the paper.

**Rel in the Real World.** Today, Rel is in use by several large enterprises to build end-to-end applications that include fraud detection, taxation, supply chain management, and others. The entire business logic for these applications is modeled in Rel, leveraging its support for programming in the large. Applications developed in Rel have run significantly faster and scaled to much larger data sets with drastically smaller code bases, when compared to the legacy applications that they replaced (up to 95% code reduction). We see this as a clear sign that the foundational concepts behind Rel deliver.

**Models and Languages for Enterprise AI.** Rel is aimed to program intelligent applications, leveraging a variety of AI techniques and reasoners to improve and automate decision making. The challenge, and the biggest cost here is to tame the highly complex mix of disparate technology stacks (e.g., for transactional, analytical, planning, graph, predictive, and prescriptive reasoning), each with its own data management methods and programming paradigm.

For example, assume that we want to answer the query *"How can we price our items to maximize income from sales in western states in the next quarter?"* Answering it comes with many challenges. First of all, it requires a combination of predictive reasoning (for predicting sales in the next quarter depending on pricing), solvers (to maximize income), and database querying. Second, the question requires precise semantics: how does the enterprise define profits, what is the date range for the next quarter, which states do we consider to be the western region? So, to express this query — whether by a human or by an agent — we need to provide the semantics of the concepts the query uses. Rel is designed to ease the burden of building such a semantic layer, and to ease writing programs that combine these different technologies. We come back to this in Section 6.4.

Furthermore, we strongly advocate for full normalization of data. Not only does it avoid semantically troublesome features such as nulls and duplicates, but it also represents data in the shape of a relational knowledge graph. This makes definitions required by the semantic layer particularly easy, and is a key reason why we advocate for the data-centric approach based on a declarative language and full normalization as a foundation for enterprise AI.

**Design Principles of Rel.** Codd [8, 9, 11, 12] introduced three main concepts: a data model based on $n$-ary relations, a normal form for database relations, and a universal (sub)language for querying database relations. We revisit these in varying depth: (1) we firmly stick to the tried and trusted relational data model, (2) we discuss data modeling and normal forms in Section 2, and (3) Rel, our proposed relational programming language, in Sections 3–6.

## 2. DATA MODELING

In this section, we explain the data modeling principles that we recommend and will help build robust applications.

**Pure Relational Model.** In the relational model, the $n$-tuples in each relation represent facts about the domain being modeled. Since a fact can neither hold multiple times, nor hold partially, the pure relational model [8] has neither nulls, nor multiplicities (bags), i.e., it is based on two-valued predicate logic and set semantics. We take this one step further: tuples represent facts that are *indivisible*. A complex fact like *"Edgar was born on 19 August 1923 in Underhill"* is better thought of as two indivisible facts, *"Edgar was born on 19 August 1923"* and *"Edgar was born in Underhill"*. Rel shares this perspective with fact-based modeling [18], inspired by the same work referenced by Codd [11]. As facts involve real-world concepts rather than database constants (*things, not strings*), tuples should store unique, context-independent representations of these concepts. In the above example, *Underhill* is the area on the Isle of Portland in Dorset (UK), not the town in Wisconsin nor the travel name of Frodo Baggins; the database should be able to distinguish between them all. The combination of indivisibility of facts and the *things, not strings* paradigm gives rise to Rel's *graph normal form (GNF)*, as we explain below.

**Graph Normal Form.** GNF relies on a conceptual model that distinguishes between *entities* (products, orders, etc.) and *values* (integers, dates, etc.). In relations, values are represented by themselves as usual, whereas entities are represented by internal *identifiers*. In GNF, identifiers are disjoint from values, and every entity in the database is represented by an identifier that is unique within the entire database. So, GNF does not allow disjoint concepts, such as product and order, to have the same identifier in the database. We call this the *unique identifier property*.

In summary, *graph normal form (GNF)* comprises the following conditions:

(1) for each $k$-ary relation $R$,
   - all $k$ columns of $R$ are the key, or
   - the first $k-1$ columns of $R$ are the key for $R$;

(2) the unique identifier property holds.

Condition (1) captures indivisibility of facts by requiring each relation to be fully normalized. Incidentally, notice that this notion basically coincides with *sixth normal form (6NF)* [14]. On the other hand, condition (2) captures the *things, not strings* paradigm by providing each entity with its unique identifier. For example, information about people could be represented using a GNF schema with relations `person(p)`, `birthDate(p,d)`, `height(p,h)`, `mother(p,m)`, `father(p,f)`, `dog(x)`, `cat(x)`, `hasPet(p,x)`, where `person(p)` tells us that `p` is a person, etc. We always define predicates such that the ordering of their tuples is consistent with how the predicate is read: `birthDate(p,d)` means that the birth date of `p` is `d`; `mother(p,m)` means that the mother of `p` is `m`, etc. Note that in Rel attributes of relations are not associated with names: they are referred by their position only. In this example we use unary relations to identify entities, thus effectively enforcing the unique identifier property.

**No nulls, no bags.** Under GNF (and 6NF), relations *do not contain nulls* (instead of using a null in the non-key column, we simply omit the whole tuple), and are *sets* (keys cannot repeat). This brings us back to Codd's original relational model; nulls and bags were introduced later for reasons that were pragmatic *at the time*, but caused problems further down the road. Nulls are known as one of the most problematic features of relational databases [13], with often unclear or misunderstood semantics [26], and a source

of many programming errors [6]. Recursion — essential for programming in the large — when coupled with bag semantics quickly leads to very large or even infinite multiplicities, and undecidability of query optimization [16, 19, 21]. Bag semantics is not needed for aggregation either, as we shall explain in Section 5. These problems disappear when we adopt *"no nulls/no bags"*, justifying it from the language design perspective. Furthermore, both set semantics and the absence of nulls restore classical logical equivalences [23, 22], thereby giving the query optimizer more headroom to do its work. Other benefits of GNF such as semantic stability and support for temporal features are discussed in [2].

## 3. INTRODUCTION TO REL

The starting point of Rel is Datalog with first-order formulas in rule bodies. We assume a basic level of familiarity with the relational data model [24, 3]. Throughout the paper, we will base our examples on the data in Figure 1, which is fully normalized, as we require in GNF (Section 2). The unique identifier property for people is enforced by the unary relation `person`, which (in the full schema) is disjoint from `dog` or `cat`. For the sake of readability, in the paper we do use strings to refer to these entities.

A Rel program is a set of rules, the most basic of which has the form

```
def RName( VariableList ) : RelExpression
```

A rule is structured like a Datalog rule: `RName` is a relation name, *VariableList* is a list of variables, and *RelExpression* is an expression that evaluates to a result that gives meaning to the variables in *VariableList*. For example, using the schema in Section 2, consider the rule

```
def parent(x,y) : mother(x,y) or father(x,y)
```

which defines `parent` as a new relation that is derived from `mother` and `father`, containing all pairs $\langle c, p\rangle$ such that $p$ is the mother or father of $c$. For example, in the instance from Figure 1 we have that the tuple ⟨"John", "Emma"⟩ belongs to the `mother` relation, while the tuple ⟨"John", "Joe"⟩ belongs to the `father` relation, meaning that both tuples will be added to `parent`. Throughout, we write tuples using angular brackets (for example, ⟨"John"⟩ is a unary tuple, and ⟨"John", "Emma"⟩ is a binary tuple).

Rule bodies can use full first-order formulas including both existential and universal quantification, as well as all Boolean operations. In the example

```
def sibling(x,y) :
  exists ((p) | parent(x,p) and parent(y,p) and x != y)
```

we say that $x$ and $y$ are siblings if they have a common parent and they are different. If we consider the database from Figure 1 we can see that "Kate" and "John" are siblings, since they have the same parent. Likewise, with the rule

```
def hasNoCats(p) : person(p) and
        forall ((x) | hasPet(p,x) implies not cat(x))
```

we say that $p$ is in `hasNoCats` if $p$ is a person and none of their pets are cats. Over the database of Figure 1 all people except "Emma" are returned, since one of her pets is a cat.

As in Datalog, rules can be recursive. For example, if we wish to express the ancestor relationship between people, we can use the following program consisting of two rules:

```
def ancestor(x,y) : parent(x,y)
def ancestor(x,y) :
  exists ((z) | parent(x,z) and ancestor(z,y))
```

The first rule states that each *parent* is also an *ancestor*, while the second rule states that each parent of an ancestor is also an ancestor. For instance, in the example of Figure 1 we have that ⟨"Kate", "Judith"⟩, ⟨"Kate", "Anne"⟩ and ⟨"Kate", "Emma"⟩ all belong to `ancestor`, since the mother/parent of "Kate" is "Emma", the mother of "Emma" is "Anne" and the mother of "Anne" is "Judith".

Although the program above is fairly easy to understand, this is much less so for general recursive programs that involve negation in rule bodies. For this reason, Rel adopts partial fixpoint semantics (iterate until a fixpoint is reached), as we discuss in our paper [1].

**Infinite Relations.** Rel allows using and even defining infinite relations. For example, we already used the infinite relation `!=` ("is different from") in the definition of `sibling`. Using another infinite relation `>=`, one can write

```
def youngestSibling(x) :
  forall ((y) | sibling(x,y) implies
    exists ((d1) | birthDate(x,d1) and
    exists ((d2) | birthDate(y,d2) and d1 >= d2)))
```

to say that $x$ is a youngest sibling. Referring again to the database in Figure 1, we can see that ⟨"John"⟩ is in `youngestSibling`, since his only other sibling, "Kate", is older. Note also that ⟨"Emma"⟩ is added to `youngestSibling` since she has no siblings and the universal quantification binds no `y` in the formula above. We will later see in detail how we can write this rule more conveniently as

```
def youngestSibling(x) :
  forall ((y) | sibling(x,y)
            implies birthDate[x] >= birthDate[y])
```

using *partial application*. For now, the reader can simply read `birthDate[x]` as "the birthdate of $x$".

Another natural use case for infinite relations is arithmetic. For instance, Rel uses `Int` to denote an infinite unary relation containing all integers and `add`, `multiply`, etc. for arithmetic. For example, `add(x,y,z)` means that the sum of `x` and `y` equals `z`, and similarly for the other arithmetic relations. These relations can also be written with the usual infix notation, so we can write

```
def addInverse(x,y) : x + y = 0
```

to define an (infinite) relation containing the pairs of numbers that are additive inverses of each other, or

```
def absolute(x,y) :
  (x >= 0 and y = x) or (x < 0 and y = -1 * x)
```

to define when the absolute value of `x` is `y`.

Note that the derived relations in the last two examples are infinite as well. In general, infinite relations not only arise from the use of primitive ones (such as `+` or `>` in the definition of `addInverse` and `absolute` above); they can also be obtained starting from finite relations, as in classical first-order logic, using, e.g., negation or disjunction. For example:

```
def nonHuman(x) : not person(x)
```

Infinite relations have an important role in Rel, as they allow the extension of the basic core language with libraries.

**Safety.** Being able to define infinite relations brings up so-called *safety* issues: to ensure that queries are computable, they should return only finitely many results.

Classical relational languages do not allow the definition of infinite relations to circumvent this problem. Relational calculus and SQL use *range restriction*. Specifically, variables can only range over elements in the database or those

| person |
|---|
| “Emma” |
| “John” |
| “Joe” |
| “Kate” |
| “Anne” |
| “Judith” |

| mother | |
|---|---|
| “John” | “Emma” |
| “Kate” | “Emma” |
| “Emma” | “Anne” |
| “Anne” | “Judith” |

| father | |
|---|---|
| “John” | ”Joe” |
| “Kate” | ”Joe” |

| birthDate | |
|---|---|
| “John” | 21–03–1988 |
| “Kate” | 28–06–1986 |
| “Emma” | 11–07–1957 |

| height | |
|---|---|
| “Kate” | 187 |
| “Joe” | 184 |

| hasPet | |
|---|---|
| “Emma” | “Snuffles” |
| “Joe” | “Luna” |

| dog |
|---|
| “Charlie” |
| “Luna” |

| cat |
|---|
| “Snuffles” |

Figure 1: Example database containing information about lineage. Some relations are omitted for brevity.

constructed from database entries by means of expressions, thereby rendering ranges of variables finite.

Rel takes a more flexible approach, as infinite relations are essential to the language. It allows unsafe definitions (i.e., unsafe rules within entire programs) as long as the final query is safe. For instance, the definition of `absolute` is not a safe query, because the relation is infinite. But if we add the definition `q` below, the query with output `q` is safe:

```
def q(x,y,h) :
  parent(x,y) and h=absolute[height[x] - height[y]]
```

The query outputs all triples $\langle x, y, h\rangle$ such that $x$ is parent of $y$ and $h$ is (the absolute value of) their height difference.

Since safety is an undecidable condition [3], it is impossible to distinguish all safe and unsafe queries. The Rel engine takes a conservative approach, ensuring that it never attempts to evaluate an expression that could be unsafe. To this end, the engine reasons about safety of queries using a set of rules, based on [17].

## 4. PROGRAMMING IN THE LARGE

We now discuss language features that, together with the basics from Section 3, enable Rel to do programming in the large and, crucially, give it the power to do meta-programming and define libraries. For this, we extend the core of Rel with four fundamental ingredients:

(1) tuple variables
(2) relation variables
(3) relational application, and
(4) abstraction.

Tuple and relation variables are needed for enabling library writing with a small core language: these are mechanisms for passing parameters that are more complex than individual values. Relational application and abstraction are relational analogs of lambda-abstraction and application that are ubiquitous in programming languages. In our design, we build upon ideas of Boute [5] who showed how to incorporate these basic programming language building blocks into predicate logic, or relational calculus.

**Tuple Variables.** Assume that we want to compute the cross product (Cartesian product) of two binary relations `U` and `V`, that is, the set of tuples $\langle a, b, c, d\rangle$ such that $\langle a, b\rangle \in$ `U` and $\langle c, d\rangle \in$ `V`. We can write this as:

```
def crossProductUV(a,b,c,d) : U(a,b) and V(c,d)
```

But what if `V` were ternary? We would have to write:

```
def crossProductUV(a,b,c,d,e) : U(a,b) and V(c,d,e)
```

Writing such code for all different arities is not only repetitive and error-prone, but also incomplete, because there is a maximal arity for which the code works. This is why Rel has *tuple variables*, which have trailing dots. Using tuple variables, we can define the cross product of `U` and `V` without knowing their arities:

```
def crossProductUV(x...,y...) : U(x...) and V(y...)
```

Here, we are saying that the cross product of `U` and `V` is the set of tuples `(x...,y...)` such that `x...` binds to a *tuple* in `U` and `y...` binds to a *tuple* in `V`. We note that, in general, tuple variables do not need to bind to *entire* tuples; they can bind to arbitrary-length sub-tuples (even sub-tuples of length zero).

**Relation Variables.** Being able to define `ProductUV` independently of the arities of `U` and `V` is nice, but it would be even better if we could pass `U` and `V` as parameters to a more general operator: we would like

```
crossProduct[U,V]
```

to return the cross product of `U` and `V`. We will achieve this in two steps: the first is *relation variables*, the second is *relational application*.

In Rel syntax, relation variables in the head of rules are indicated by enclosing them in curly brackets:

```
def crossProduct({A},{B},x...,y...) : A(x...) and B(y...)
```

Then, we can write `crossProduct(U,V,a,b,c,d)` to test if $\langle a, b, c, d\rangle$ is in the cross product of `U` and `V`.

Adhering to Rel's fully relational design, `crossProduct` is a relation. However, *conceptually*, it is second-order, as its first two columns contain relations instead of values:

| crossProduct | | | | | |
|---|---|---|---|---|---|
| $\{0\}$ | $\{0\}$ | 0 | 0 | | |
| $\{0\}$ | $\{1\}$ | 0 | 1 | | |
| $\{1\}$ | $\{0\}$ | 1 | 0 | | |
| ... | | | | | |
| $\{\langle 1,2\rangle, \langle 3,4\rangle\}$ | $\{\langle 5,6\rangle\}$ | 1 | 2 | 5 | 6 |
| $\{\langle 1,2\rangle, \langle 3,4\rangle\}$ | $\{\langle 5,6\rangle\}$ | 3 | 4 | 5 | 6 |
| ... | | | | | |

The relation `crossProduct` is also infinite. It has infinitely many rows, because it contains infinitely many relations in its first column alone. It also has infinitely many columns, because the relations in the first two columns do not have an upper bound on their arity. As such, the tuples in their cross product become arbitrarily long.

*Remark.* Readers familiar with the nested relational data model may also think of `crossProduct` as a nested relation. Such relations cannot be used as base relations nor query outputs, but conceptually they are necessary when relations themselves can be passed as parameters.

**Relational Application.** We can write `mother("John","Emma")` to test if ⟨“John”, “Emma”⟩ ∈ `mother`, which is the standard notation in our field for *relational atoms* [3, 24]. Another, more general way of understanding this is seeing `mother` as a Boolean function that, given two input parameters, $a$ and $b$, returns whether $\langle a, b\rangle \in$ `mother`. This principle works for all kinds of relations we have seen, including second-order ones. For example, if relation `R` contains the tuple $\langle 1, 2\rangle$, and relation `S` contains the tuple $\langle 5, 6\rangle$, then

```
crossProduct(R, S, 1, 2, 5, 6)
```

evaluates to true, because the second-order tuple $\langle R, S, 1, 2, 5, 6\rangle$ is in the `crossProduct` relation. What we just described is one form of *relational application* in Rel. We call it *full application*, because the syntax with round brackets requires that all arguments be given.

Rel also supports *partial application*, which uses square brackets instead of round ones, and returns all suffixes in a relation that are consistent with a given prefix. For example, `parent["John"]` contains $\{\langle$"Emma"$\rangle, \langle$"Joe"$\rangle\}$ for the database in Figure 1, since both $\langle$"John","Emma"$\rangle$ and $\langle$"John","Joe"$\rangle$ are in `parent`. The same principle holds for second-order relations. So,

```
crossProduct[R, S]
```

evaluates to the cross product of the relations $R$ and $S$.

In fact, since the cross product of two relations is so common in database applications, Rel has the syntactic sugar

```
(R,S)
```

to abbreviate `crossProduct[R, S]`. For example, `(mother, parent)` is the cross product of `mother` and `parent`, while `("Joe",40)` is the singleton relation $\{\langle$"Joe", 40$\rangle\}$.

We need not provide all arguments in partial application, which thus evaluates to a relation. For example, `crossProduct[{1}]` is a second-order relation that can be applied further. To illustrate this, take the expression `{crossProduct[{1}]}[cat]` using the relation `cat` from Figure 1. This will return $\{\langle 1,$ `"Snuffles"`$\rangle\}$. We can also name the expression by writing `def CP1 {crossProduct[{1}]}`, after which `CP1[cat]` returns the same result. This is a relational version of *currying* from functional programming. Indeed, any relation $R$ can be seen as a function $a \mapsto R_a$ where $R_a = \{\langle \overline{b}\rangle \mid \langle a, \overline{b}\rangle \in R\}$. By passing $a$ as a parameter to $R$, we obtain a new relation $R_a$ that takes one parameter less.

If the relation resulting from partial application has arity zero, the result is either $\{\langle\rangle\}$ (the relation with the empty tuple) or $\{\}$ (the empty relation). In Rel, these two relations encode Boolean values true and false, respectively, as usual in the relational data model [3]. Therefore, full application computes the same result as partial application if all arguments are provided.

*Expressions and Formulas.* With partial application, we can write *RelExpressions* that evaluate to relations rather than Booleans. In fact, Rel calls *Formula* any *RelExpression* that evaluates to a Boolean, and this distinction allows for some syntactic sugar. For instance, if we want to condition the evaluation of a *RelExpression* on the truth of a *Formula*, we can write

*RelExpression* `where` *Formula*

and one may verify that it is equivalent to the cross product `(`*RelExpression*`,` *Formula*`)`. Indeed, taking the cross product of any relation $R$ with $\{\langle\rangle\}$ is $R$ itself, and taking the cross product of $R$ with the empty relation is empty.

**Abstraction.** Until now we used Rel rules of the form:

`def` RName`(`*VariableList*`):` *Formula* (1)

Actually, Rel rules can also have the form:

`def` RName`[`*VariableList*`]:` *RelExpression* (2)

In both forms, the expression following `RName` is called *Abstraction* and evaluates to a relation. In general, *Abstraction* can have one of the following forms:

`{ (` *VariableList* `) :` *Formula* `}` (3a)

`{ [` *VariableList* `] :` *RelExpression* `}` (3b)

and the outer curly braces can be omitted to allow the forms (1) or (2).

Abstractions of the form (3a) are inspired by *set comprehension* from mathematics, where we write, e.g., $\{n \in \mathbb{N} : \exists m \in \mathbb{N}$ such that $n = 2m\}$ for the set of even numbers. For example,

```
{ (x,y) : father(x,"Joe") and ancestor("Joe",y) }
```

defines a binary relation linking children of "Joe" with their paternal ancestors.

Abstractions of the form (3b) use square brackets and allow an arbitrary *RelExpression* on the right. For example

```
{ [x,y] : height[y] - height[x] where parent(x,y) }
```
(4)

defines a ternary relation containing, for each child-parent pair, the amount that the parent is taller than the child. Here we iterate over all possible pairs $\langle v_x, v_y\rangle$ of values for `x` and `y` and compute the expression on the right of the colon. That is, for each pair $\langle v_x, v_y\rangle \in$ `parent` we find the tuples $\langle v_x, h_x\rangle$ and $\langle v_y, h_y\rangle$ in `height` and add $\langle v_x, v_y, h_y - h_x\rangle$ to the result. On the database of Figure 1, we would add the tuple $\langle$"Kate","Joe",$-3\rangle$ to the result. Notice that in general the expression on the right of the colon does not need to compute a single value but can evaluate to any set of tuples.

We call this operation *(relational) abstraction* to draw a parallel with functional abstraction in functional programming languages. Indeed, lambda-abstraction $\lambda x.e$ denotes a function that for argument $x$ computes $e(x)$. This function can be viewed as a relation, namely the set of pairs $(x, e(x))$. In relational abstraction, the difference is that $e$ can be an expression that produces a relation instead of a function. The semantics of the abstraction then is the set of tuples $(x, y_1, \ldots, y_k)$ with $(y_1, \ldots, y_k) \in e(x)$.

Using the syntax given in (2), relational abstraction may be used to define rules. For instance, the next rule defines `parentHeightDiff` to be the relation previously computed by (4):

```
def parentHeightDiff[x,y] :
  height[y] - height[x] where parent(x,y)
```

Let us stress that the rule above adds *triplets* to `parentHeightDiff`: the first two columns correspond to `[x,y]`, and the rest corresponds to the arity of the *RelExpression* right of the colon (which in this case is one more column).

## 5. AGGREGATION

Aggregation, a crucial component of query languages, is implemented in Rel as part of the standard library. For instance, `sum[R]` returns the sum of the elements in the last column of relation `R`. This finds justification in the underlying GNF for `R`: if we want to aggregate values in `R`, then `R` is going to be a set of key/value pairs, where the last column contains the values. The usual aggregates `sum`, `count`, `min`, `max`, `average` (and more) are available.

**Aggregation with Grouping under Set Semantics.** One often hears the argument that bag semantics is necessary for defining aggregation correctly. However, bags are only required if one looks solely at the aggregated column, and not at tuples in the relation that contain the aggregated column. We illustrate this using the schema from Section 2.

Assume that we wish to compute the average height of children of each person $p$. This is defined using relational application and abstraction:

```
def avgChildHeight[p] :
  average[(c,h) : parent(c,p) and height(c,h)]
```

The example illustrates how set semantics interacts with grouping: the argument of `average` evaluates to a set of pairs $\langle$ *child*, *height* $\rangle$, which makes multiple same-height children of a person $p$ all contribute to the average computed over the last column. It also illustrates how grouping by persons $p$ is done using relational abstraction on `[p]`. Note that if we had only kept the heights in the argument of `average`, we would have computed something different:

```
def notTheAvgChildHeight[p] :
average[(h) : exists((c) | parent(c,p) and height(c,h))]
```

Indeed, here, for each $p$, we define the set $\{h : h$ is a height of some child of $p\}$ and take the average of the values in the set. As such, this definition counts two children with the same height only once due to set semantics.

Therefore we can conclude that *aggregation works with set semantics.* With the right language design, bags are not needed to cleanly define aggregation.

**Aggregation and Linear Algebra.** The design of aggregation and partial application in Rel makes it very natural to write code for linear algebra. Let us model vectors as binary relations: the first column holds a position, and the second holds the value at that position. Matrices are encoded as ternary relations: the first two columns encode a position (row and column), and the third the value. Two such examples of encoding a vector $\mathbf{v}$ of length 4 and a $2 \times 2$-matrix $\mathbf{M}$ are shown below.

$$\underbrace{\begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix}}_{\mathbf{v}} \Rightarrow \begin{array}{|cc|}\hline 1 & v_1 \\ 2 & v_2 \\ 3 & v_3 \\ 4 & v_4 \\ \hline \end{array}_{V} \qquad \underbrace{\begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}}_{\mathbf{M}} \Rightarrow \begin{array}{|ccc|}\hline 1 & 1 & m_{11} \\ 1 & 2 & m_{12} \\ 2 & 1 & m_{21} \\ 2 & 2 & m_{22} \\ \hline \end{array}_{M}$$

The same principle works for tensors: $k$-dimensional tensors are encoded as $(k+1)$-ary relations, where the first $k$ columns encode the tensor coordinates, and the last one the value.

Using `sum`, we can now define the sum of the elements of row $i$ as:

```
def rowSum[{A},i] : {sum[[j] : A[i,j]]}
```

It is not a coincidence that the code for `rowSum[A,i]` looks similar to the mathematical expression $\sum_j a_{ij}$. The ranges of `i` and `j` are guarded by the first and second column of $A$, respectively. Hence, for each value of $i$ that appears in the first column of relation $A$, the subexpression `[j] : A[i,j]` evaluates to the binary relation $\{\langle j, a_{ij}\rangle \mid j$ appears in the second column of $A\}$. On the relation

$$\begin{array}{|ccc|}\hline 1 & 1 & 1 \\ 1 & 2 & 2 \\ 2 & 1 & 0 \\ 2 & 2 & 4 \\ \hline \end{array}_{M} \qquad \left(\text{representing the matrix } \begin{pmatrix} 1 & 2 \\ 0 & 4 \end{pmatrix}\right)$$

`rowSum[M,1]` returns 3. We do not have to provide the row numbers: the expression `rowSum[M]` returns $\{\langle 1, 3\rangle, \langle 2, 4\rangle\}$.

*Missing Values.* It is interesting to observe that `rowSum` computes exactly the same results if the tuple $\langle 2, 1, 0\rangle$ were missing from `M`. This means that representing sparse matrices is automatically supported, although one has to be careful not to exclude too many values (e.g., inferring the dimensions of a matrix that only consists of zero-entries is not possible if all tuples are missing). We discuss more complex examples in linear algebra, such as matrix multiplication and PageRank in Section 6.

# 6. CODE EXAMPLES

We now discuss a few slightly larger code examples that illustrate several prominent features of Rel programming. For the sake of compactness, this section's code will make use of *wildcard* (_), which in Rel, as in Datalog, denotes an anonymous existentially quantified variable, e.g.,

```
(y) : R(_,y,_...)
```

is equivalent to `(y) : exists ((x,z...) | R(x,y,z...))` and evaluates to the second column of relation `R`.

## 6.1 Relational and Linear Algebra

Notation for linear algebra (LA) and relational algebra (RA) as well as languages such as APL [20] and FP [4] allows writing programs using a set of generally useful primitives and avoiding named variables, a style called *point-free* or *tacit programming*. The point-free style is also quite popular in business intelligence tools. Rel supports point-free style via libraries rather than language extensions, which we now demonstrate with libraries for RA and LA.

**Relational Algebra.** A simple example of relations defined in the library are familiar relational algebra operators. Cartesian product is defined as the `crossProduct` relation that we already discussed in Section 4. We also mentioned there that Rel uses the infix notation `(A,B)` to denote the Cartesian product of `A` and `B`.

The union of two relations can also be defined in the library as follows:

```
def union({A},{B},x...) : A(x...) or B(x...)
```

Just like product, union has an infix shorthand: `{A; B}`. This way we can build arbitrary relations from constants; e.g.,

```
{(1,2,3) ; (4,5,6) ; (7,8,9)}
```

evaluates to $\{\langle 1, 2, 3\rangle, \langle 4, 5, 6\rangle, \langle 7, 8, 9\rangle\}$. The remaining set operator of relational algebra — difference — is similarly defined by the following code:

```
def minus({A},{B},x...) : A(x...) and not B(x...)
```

A selection operator simply takes a relation and a condition — which would be an infinite relation — and returns their intersection:

```
def select({A},{Cond},x...) : A(x...) and Cond(x...)
```

Consider, for example, a relational algebra expression $\sigma_{1=2}(R \times S) \cup B$. To express this in Rel in point-free style, we first need an infinite set of tuples whose first and second components are equal:

```
def cond12(x1,x2,x...) : {x1=x2}
```

and then define the entire expression as follows:

```
union[select[crossProduct[R,S],cond12],B]
```

Projection can be easily expressed in Rel using abstraction. For instance, the projection of a relation on the first and third attribute may be expressed by the following code:

```
def proj13 ({A},x,y) : A(x,_,y,_...)
```

**Linear Algebra.** Recall the relational representation of vectors, matrices, and tensors from Section 5. We now dive a bit deeper into linear algebra and look at slightly more complex examples. Given two vectors $\mathbf{u} = (u_1, \ldots, u_n)$ and $\mathbf{v} = (v_1, \ldots, v_n)$, their scalar product is $\mathbf{u} \cdot \mathbf{v} = \sum_k u_i v_i$. Rel's definition below mimics the mathematical definition:

```
def scalarProd[{U},{V}] : {sum[[k] : U[k]*V[k]]}
```

The range of `k` is guarded by the first columns of $U$ and $V$, hence the subexpression `[k] : U[k]*V[k]` evaluates to the

binary relation $\{\langle i, u_i v_i\rangle \mid i \in \{1,\ldots,n\}\}$. Recall that the `sum` aggregate computes the sum of the values in the last column of its argument (see Section 5). Hence this rule indeed computes $\sum_k u_i v_i = \mathbf{u} \cdot \mathbf{v}$.

The same approach makes it easy to define matrix multiplication. Recall that if $\mathbf{M} = \mathbf{A} \cdot \mathbf{B}$, then its entry $m_{ij}$ in the $i$th row and $j$th column is $\sum_k a_{ik} \cdot b_{kj}$, as reflected in the following Rel definition:

```
def matProd[{A},{B},i,j] : {sum[[k] : A[i,k]*B[k,j]]}
```

Similarly, $\mathbf{A} \cdot \mathbf{v}$ for a matrix $\mathbf{A}$ and a vector $\mathbf{v}$ is defined in Rel by what follows:

```
def matVectProd[{A},{V},i] : {sum[[k] : A[i,k]*V[k]]}
```

An advantage of point-free code is that it is more robust under changes of the underlying data. For instance, `Union[R,S]` and `MatProd[A,B]`, work independently of the arities of the relations `R` and `S` or the dimensions of the matrices `A` and `B`.

## 6.2 Graph Algorithms

**All Pairs Shortest Paths.** Next, we demonstrate code for the all pairs shortest path problem. The following code expects two relation variables `V` and `E`, representing the finite set of vertices and edges (which are pairs of vertices) of a graph.

```
def APSP({V},{E},x,y,0) : V(x) and V(y) and x = y
def APSP({V},{E},x,y,i) :
  exists ((z in V) | E(x,z) and APSP(V,E,z,y,i-1)) and
  not exists ((j in Int) | j < i and APSP(V,E,x,y,j))
```

The first rule states that the shortest path from a node to itself has length 0. The second rule states that the shortest paths from `x` to `y` have length $i$ if there exists an out-neighbor `z` of `x` such that the shortest paths from `z` to `y` have length $i-1$ and we have not already discovered paths shorter than $i$ from `x` to `y`. Although we use quantification over the entire relation `Int`, the Rel engine will not loop over all integers to compute the query.

We can also define `APSP` using aggregation and abstraction, as shown below.

```
def APSP({V},{E},x,y,0) : V(x) and V(y) and x = y
def APSP[{V},{E},x,y] :
  min[(j) : exists((z) | E(x,z) and APSP(V,E,z,y,j-1))]
```

**PageRank.** Using the vector and matrix encoding from Section 5 and basic linear algebra functions from Section 6.1, we now write (a simplified version of) the PageRank algorithm in Rel. We also illustrate how a Rel program can perform a number of steps until a stopping condition is met.

```
def vector[d,i]  : 1.0/d where range(1,d,1,i)
def dimension[{A}] : max[(k) : A(k,_,_) or A(_,k,_)]
def delta[{U},{V}] : max[[k] : absolute[U[k] - V[k]]]

def next[{A},{V}] : {matVectProd[A,V]}
def stop({A},{V}) : {delta[next[A,V],V] < 0.005}

def pageRank[{A}] :
  {vector[dimension[A]] where empty(pageRank[A])}
def pageRank[{A}] :
  {next[A,pageRank[A]] where not empty(pageRank[A])
                        and not stop(A,pageRank[A])}
def pageRank[{A}] : {pageRank[A] where
  not empty(pageRank[A]) and stop(A,pageRank[A])}
```

Here, `absolute` is defined as in Section 3, `empty` is the emptiness test given by

```
def empty({R}) : not exists((x...) | R(x...))
```

and the predicate `range(k,d,1,i)` is true for $i = k, k+1, \ldots, d$. Given an input graph, if `M` is the usual input matrix of the PageRank algorithm whose entry $(i,j)$ is $1/k$, where $k$ is the $j$-th vertex's out-degree, then `pageRank[M]` iteratively computes PageRank of the graph, until the delta between two consecutive iterations is at most 0.005.

## 6.3 Querying Property Graphs

Property graphs, a popular graph data model, can be easily represented relationally. A possible GNF encoding is as follows: we have relations `node(u)` and `edge(e)` for the graph's nodes and edges, as well as `src(e,u)`, `tgt(e,v)` to indicate the source and target node of each edge $e$. Simplifying for the examples here, assume that nodes and edges have one property `V`. This is encoded by a relation `V(x,w)` that pairs each node or edge `x` with a value `w`.

To start with, we define the reachability query that can be applied to an arbitrary binary relation `E`:

```
def reachable({E},x,y) :
  E(x,y) or exists((z) | E(x,z) and reachable(E,z,y))
```

Assume that we want to find node pairs $(u,v)$ of a property graph such that there is a path from $u$ to $v$ with increasing values on nodes. We start with a binary relation `goodNodePair` of nodes $(u,v)$ connected by an edge so that the value in $v$ is greater than the value in $u$:

```
def goodNodePair (u, v) : exists( (e in edge) |
    src(e,u) and tgt(e,v) and V[u] < V[v] )
```

The expression `reachable[goodNodePair]` gives us the desired node pairs. Assume now that we want to find node pairs $(u,v)$ such that there is a path from $u$ to $v$ with increasing values on *edges*. This seemingly very similar query is however not naturally expressible with pattern matching facilities of existing graph query languages [15]. At the same time, it is very easy to define in Rel:

```
def goodEdgePair (e1,e2) : edge(e1) and edge(e2)
      and tgt[e1] = src[e2] and V[e1] < V[e2]
```

Now, the expression `reachable[goodEdgePair]` gives us edge pairs $(e_1, e_2)$ such that there is a path from the source $u$ of $e_1$ to the target $v$ of $e_2$ with increasing values on edges. To get node pairs $(u,v)$ themselves, we can use:

```
def incValOnEdges(u,v) :
    exists((e1,e2) | reachable(goodEdgePair,e1,e2)
                  and src(e1,u) and tgt(e2,v))
```

## 6.4 Semantics and Business Logic

In the introduction we stated that Rel can express complete business logic by bringing semantics close to the data, and making use of various reasoners. Let us turn this slogan into something more tangible and consider the question: *"How can we price our items to maximize income from sales in western states in the next quarter?"* This is not a traditional database query for at least three different reasons. First, it refers to concepts that are not yet completely defined. For example, which states are Western? What quarter is next? Second, it needs data that we do not yet have in the database: sales for the next quarter. This will be obtained by a *predictive reasoner* that we can invoke. Third, how do we optimize? This will be done by a *prescriptive reasoner*, or solver. Finally, answering our question is a multi-step process involving database queries for defining semantics, as well as data that will be used by reasoners — all this data comes in shape of relations produced and consumed by Rel.

Assume (simplified for the paper) that we have base relations `itemPrice(item, state, quarter, price)`,

`sales(item, day, state, numItems)`, `inventory(item, day, state, numItems)` of historical data. Semantic concepts used in queries are relations defined in Rel, e.g.,

```
def dayQuarter(day,quarter) : ...
def stateRegion {...}
```

These are simple Rel queries the first of which maps a day to a quarter based on factors such as the beginning of the fiscal year, and the second contains pairs such as `("CA","Western"),("NY","Eastern")` etc.

So how to find the prices that maximize our income from sales? For this example, we assume a simplified model where the number of items $i$ sold in a given quarter is $s_i = m_i - \sum_j e_{ij} p_j$, where $m_i$ is the inventory of item $i$ at the beginning of the quarter and $p_i$ is its price, and $e_{ij}$ is the *elasticity coefficient* giving how the price of item $j$ affects sales of item $i$. Based on historical data, we will be predicting elasticity coefficients for the next quarter. We will then use them to find optimal prices.

First we need the data from past quarters to feed to our predictive reasoner. This is obtained by querying the database. For example, to obtain the relevant historical inventories $m_i$ (in the Western states) at the beginning of each quarter, we write:

```
def invW[item,quarter] : sum[ [day,state] :
  inventory[item,day,state] where firstDay(day,quarter)
  and stateRegion[state]="Western"]
```

where `firstDay(d,q)` says that `d` is the first day of quarter `q`. Likewise, we define a relation `totalSalesW(item,amount)` of total sales $s_i$ in the Western region (similarly obtained by aggregation). The historical values of $p_j$ come from `itemPrice`. Notice that by creating semantic definitions for concepts like `stateRegion`, we can continue using exact same queries even when the semantics changes, say if `"HI"` is added to Western states, or the beginning of the fiscal year is modified.

Second, these relations are passed as relational parameters to a predictive reasoner that computes a matrix `el` for the elasticity coefficients $e_{ij}$. The matrix is represented as a relation `el[i,j,coeff]`, as in Section 5.

Third, we must now find prices $p_i$ that maximize total sales income in the next quarter, which is $\sum_i s_i p_i = \sum_i (m_i p_i - \sum_j e_{ij} p_i p_j)$. We obtain the $m_i$'s (for the next quarter `next_q`) using the expression `{[i]:invW[i,next_q]}`, and pass it together with `el` to a quadratic programming solver (a Rel library). It returns the optimal prices $p_i$ in the form of a relation `result(item,optimalprice)`.

Thus, using appropriate libraries in Rel, one can program the entire application end to end in a purely relational fashion, with all the important concepts: semantic definitions that stay close to data, parameters for predictive and prescriptive reasoners, and their outputs, being relations defined in Rel. By consistently applying a design based on a small number of powerful primitives, Rel is designed to make this programming task easy for both human programmers and GenAI.

## Acknowledgments

We are deeply indebted to Martin Bravenboer, who is one of the original designers of Rel and to the RelationalAI product and field engineering teams for being early users of Rel and providing invaluable feedback that helped improve it.

## 7. REFERENCES

[1] M. Aref, P. Guagliardo, G. Kastrinis, L. Libkin, V. Marsault, W. Martens, M. McGrath, F. Murlak, N. Nystrom, L. Peterfreund, A. Rogers, C. Sirangelo, D. Vrgoč, D. Zhao, and A. Zreika. Rel: A programming language for relational data. In *SIGMOD*, pages 283–296. ACM, 2025.

[2] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382, 2015.

[3] M. Arenas, P. Barceló, L. Libkin, W. Martens, and A. Pieris. *Database Theory*. Open source at `https://github.com/pdm-book/community`, 2022.

[4] J. W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[5] R. T. Boute. Functional declarative language design and predicate calculus: a practical approach. *ACM Trans. Program. Lang. Syst.*, 27(5):988–1047, 2005.

[6] S. Brass and C. Goldberg. Semantic errors in SQL queries: A quite complete list. *J. Syst. Softw.*, 79(5):630–644, 2006.

[7] S. Cass. The top programming languages 2024. `https://spectrum.ieee.org/top-programming-languages-2024`.

[8] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[9] E. F. Codd. A database sublanguage founded on the relational calculus. In *ACM-SIGFIDET Workshop on Data Description 1971*, pages 35–68. ACM, 1971.

[10] E. F. Codd. Relational completeness of data base sublanguages. *IBM Research Report*, RJ987, 1972.

[11] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.

[12] E. F. Codd. Relational database: A practical foundation for productivity. *Commun. ACM*, 25(2):109–117, 1982.

[13] C. J. Date. *Database Technology: Nulls Considered Harmful*. Technics Publications, 2024.

[14] C. J. Date, H. Darwen, and N. A. Lorentzos. *Temporal data and the relational model*. Elsevier, 2002.

[15] A. Gheerbrant, L. Libkin, L. Peterfreund, and A. Rogova. GQL and SQL/PGQ: theoretical models and expressive power. *Proc. VLDB Endow.*, 18(6):1798–1810, 2025.

[16] T. J. Green. Bag semantics. In *Encyclopedia of Database Systems, Second Edition*. 2018.

[17] P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund, and C. Sirangelo. Queries with external predicates. In *ICDT*, pages 22:1–22:20, 2025.

[18] T. A. Halpin and T. Morgan. *Information modeling and relational databases*. Morgan Kaufmann, 2008.

[19] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Trans. Database Syst.*, 20(3):288–324, 1995.

[20] K. E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., USA, 1962.

[21] T. S. Jayram, P. G. Kolaitis, and E. Vee. The containment problem for REAL conjunctive queries with inequalities. In *PODS*, pages 80–89, 2006.

[22] L. Libkin and L. Peterfreund. SQL nulls and two-valued logic. In *PODS*, pages 11–20, 2023.

[23] J. Marcinkowski and P. Ostropolski-Nalewaja. Bag semantics query containment: The CQ vs. UCQ case and other stories. *PACMMOD*, 3(5):1–24, 2025.

[24] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.

[25] G. L. Steele. Growing a language. *High. Order Symb. Comput.*, 12(3):221–236, 1999.

[26] E. Toussaint, P. Guagliardo, L. Libkin, and J. Sequeda. Troubles with nulls, views from the users. *Proc. VLDB Endow.*, 15(11):2613–2625, 2022.