An Extended Tutorial and Vocabulary for Relational Language Design in an Era of AI-Assisted Query Generation

Wolfgang Gatterbauer
Northeastern University
Boston, MA, USA

ABSTRACT

Relational query languages have been studied and used for more than 50 years, with SQL dominant in practice. Today, queries are increasingly generated by machines and read by humans. At the same time, the landscape also includes dataframe, pipeline, logical, functional, graph, and relational programming notations. These developments invite two related questions beyond expressive power: *which relational structures do languages make explicit*, and how well can notation *support users in reading and revising queries*?

This 3-hour tutorial extends an earlier SIGMOD'26 tutorial in three directions: recursive and path queries (connecting relational and graph query languages), nested relational data, and relational languages for problems beyond PTIME. Rather than beginning from formal definitions, we start from example queries and compare how different languages express the same intent. To compare recurring structure across notations, we use Abstract Relational Calculus (ARC) and Relational Diagrams as reference representations.

From these examples, we develop a vocabulary for relational language design, including *information need*, *query mapping*, *relational pattern structure*, *relational pattern denotation*, and *semantic conventions*. Participants will leave with a framework for comparing existing and future relational languages, a precise vocabulary for articulating design trade-offs, and a concrete set of examples connecting classical database languages with alternative proposals.

PVLDB Reference Format:
Wolfgang Gatterbauer. An Extended Tutorial and Vocabulary for Relational Language Design in an Era of AI-Assisted Query Generation. PVLDB, 19(12): 4932-4938, 2026.
doi:10.14778/3827998.3828153

PVLDB Artifact Availability:
The slides from the tutorial will be made available on the tutorial web page: https://northeastern-datalab.github.io/relational-language-tutorial/.

1 INTRODUCTION

Donald Knuth is quoted as saying, "*I think of a programming language as a tool to convert a programmer's mental images into precise operations that a machine can perform*" [56]. In the same spirit, the database community has primarily viewed relational query languages as specifications for translating information needs into executable commands. This perspective has helped make *expressive power* a central criterion for comparing and designing relational languages [19]. Yet *what a language can say* is only one aspect of language design. Another one is *how effectively* it lets us formulate, understand, and revise what we want to say.

In his 1979 Turing Award lecture "*Notation as a Tool of Thought*" [43], Iverson cites Whitehead via Cajori: "*By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases (our) mental power...*" Good notation can make relational structure explicit (e.g., a join path) so that readers can perceive it directly rather than having to reconstruct and retain it in their minds as they read a query. Yet this view of notation design for "cognition enhancement" has not received much attention in relational language design. It echoes the idea of *linguistic relativity*: the languages we use can shape how we represent and reason about problems [74].

This perspective is especially timely as generative AI and large language models (LLMs) are changing how users interact with data. Queries are no longer only instructions for machines. Increasingly, they are shared working representations through which *humans can understand what machines have done or intend to do*: machines generate and transform queries, while humans inspect, interpret, verify, and revise them [31, Fig. 2]. As noted in [5, 31, 32], this shifts attention from human query composition toward machine query generation and *human query interpretation*. When machine-generated queries can be wrong or misleading, making them easy to read, verify, and revise becomes crucial.

At the same time, there is renewed interest in relational language design. A number of recent efforts propose extensions or alternatives to SQL (including [1, 7, 57, 66]) and differ in how they present relational structure: through nested or pipelined composition, named or positional access to relation components, and tuple or domain variables. Such choices affect which structural patterns are easy to recognize and which revisions are easy to make. Yet comparing these choices systematically requires a common vocabulary that *separates notation from relational structure and semantics*.

What we need, more broadly, is systematic research about how design choices in relational languages affect how humans and machines think and reason. A necessary ingredient is a framework and vocabulary for discussing *how different relational languages realize the same relational intent (or query mapping) while making different structures, assumptions, and notations visible* to humans and machines. Without a precise vocabulary, debates about relational languages tend to confuse surface syntax and semantic conventions.

This tutorial develops and applies such a relational language design framework (Fig. 1) that distinguishes and disentangles several concepts that discussions of query languages often conflate. Rather than beginning with formal definitions, we start from a fixed set of queries and compare their realizations across various

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828153

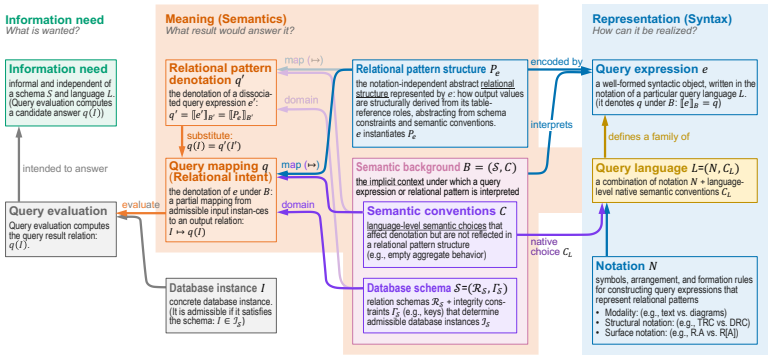

Figure 1: Relationships between concepts in the vocabulary of our relational language design framework that we will use.

Empl

eid	k
a	1
b	2
c	3

Sales

eid	val	date
a	10	1/1/26
a	20	1/2/26
b	40	1/1/26

(a) $\mathcal{S}_1, I_1$

```
select E.eid,
 (select sum(S.val) as sm
  from Sales S
  where E.eid = S.eid)
from Empl E
```

(b) e_1 **(SQL)**

```
Q(e, sum v: {Sales(e,v,_)}) :- Empl(e,_).
```

(c) e_2 **(Soufflé)**

eid	sm
a	30
b	40
c	NULL

(d) $q_{e_1,(\mathcal{S}_1,C_{\mathbf{SQL}})}(I_1)$

eid	sm
a	30
b	40
c	0

(e) $q_{e_2,(\mathcal{S}_1,C_{\mathbf{Soufflé}})}(I_1)$

Figure 2: A database schema and instance with underlined attributes forming primary keys (a), two expressions with similar conceptual evaluation strategies intended to answer the same information need (b, c), and their different query results under the languages' native semantic conventions (d, e).

relational languages. The examples expose differences in relational pattern, semantic conventions, and structural notation before these distinctions are formalized. To make recurring relational patterns explicit, we use Abstract Relational Calculus (ARC) [32], together with Relational Diagrams [29, 30], as common reference representations. Our goal is to survey languages, clarify the design space, make the comparison concrete through a set of recurring examples, and develop a precise vocabulary for relational language design to enable clear communication.

This 3-hour tutorial extends an earlier SIGMOD 2026 tutorial [28] in three directions: (1) recursion and path queries (connecting relational and graph query languages), (2) nested relational data, and (3) relational formalisms for problems beyond PTIME.

2 AN ILLUSTRATIVE END-TO-END EXAMPLE

> *"A good stock of examples, as large as possible, is indispensable for a thorough understanding of any concept, and when I want to learn something new, I make it my first job to build one."*
>
> — Paul Halmos [39, p. 63]

Information need, query expressions, and query mappings. An *information need* η describes the information sought in a given context, often informally, such as "*Return the sum of each employee's sales.*" It does not yet fix a database schema or query language, nor necessarily all semantic details, such as how employees without sales should be treated.

Figure 2a shows a database instance I_1 over a database schema $\mathcal{S}_1$ with relation names Empl and Sales, whose current relation values $I_1(\text{Empl})$ and $I_1(\text{Sales})$ each contain three tuples. The schema also specifies integrity constraints, which restrict the admissible database instances: the primary keys are {eid} for Empl and {eid, date} for Sales, and Sales.val is `NOT NULL`.

Figures 2b and 2c show two *query expressions* for answering η: an SQL expression e_1 and a Soufflé expression e_2. To interpret a query expression e, we fix a *semantic background* $B = (\mathcal{S}, C)$ where the database schema $\mathcal{S}$ determines the admissible input instances and the *semantic conventions* C determine how the constructs in e are interpreted. The resulting *query mapping* is the denotation

$$q_{e,B} \coloneqq [\![e]\!]_B,$$

which we write as q if e and B are clear from context. It is a possibly partial *function* from admissible instances of $\mathcal{S}$ to output relation values, and $q_{e,B}(I)$ is the result of evaluating e on I under B.[1]

A query language $L = (N, C_L)$ consists of a notation N together with its native semantic conventions C_L.[2] An expression written in L is therefore interpreted natively using $C = C_L$, although we may also interpret it under another compatible choice of C.

Semantic conventions affect query mappings. Despite their different surface notations, the *conceptual evaluation strategies* of e_1 and e_2 share the same relational core: each takes an employee tuple from Empl, selects the matching Sales tuples on eid, sums their val values, and returns the employee identifier together with the sum.[3] For employees a and b, the matching sales values sum to 30 and 40 in Fig. 2a. Employee c, however, has no matching sale, so the collection of values being aggregated is empty. The native semantic

[1] The earlier SIGMOD'26 tutorial [28] used "query intent" for what we now call *information need* and "relational intent" for *query mapping*. The original terminology was meant to contrast an informal information need with its formal realization as a mapping, but feedback showed that the two uses of "intent" were confusing. "Query mapping" now also follows the terminology of Abiteboul et al. [4, Ch. 4] and is used in addition to "relational intent." "Information need" is preferred over "user intent" because it does not assume a human user.

[2] *Notation* is the externally represented scheme of symbols, arrangement conventions, and formation rules by which query expressions are constructed and recognized. We distinguish three components: *modality* (the external representational form in which notation is realized, such as text or diagrams), and two types of notational design choices, *structural notation* and *surface notation*, which are either modality-independent or modality-dependent.

[3] For a description of SQL's conceptual evaluation strategy, see [61, Sect. 5.2 and 5.4]. For the semantics of Soufflé aggregates, see the official documentation [2], which describes how an aggregate depending on an outer-scope variable produces a result for each valid assignment of that variable.

conventions of SQL and Soufflé differ on this case:

$$\mathrm{sum}_{C_{\mathrm{SQL}}}(\varnothing) = \mathtt{NULL}, \qquad \mathrm{sum}_{C_{\mathrm{Soufflé}}}(\varnothing) = 0.$$

Let $B_{1,\mathrm{SQL}} \coloneqq (\mathcal{S}_1, C_{\mathrm{SQL}})$ and $B_{1,\mathrm{Soufflé}} \coloneqq (\mathcal{S}_1, C_{\mathrm{Soufflé}})$. The two expressions therefore give different query results on the same instance I_1:

$$q_{e_1,B_{1,\mathrm{SQL}}}(I_1) \neq q_{e_2,B_{1,\mathrm{Soufflé}}}(I_1),$$

or in short, $q_1(I_1) \neq q_2(I_1)$. Concretely, e_1 returns NULL for employee c, whereas e_2 returns 0. Since the two functions disagree on an admissible input, they are different:

$$q_{e_1,B_{1,\mathrm{SQL}}} \neq q_{e_2,B_{1,\mathrm{Soufflé}}}.$$

Thus, two expressions may address the same information need in different notations and have conceptual evaluation strategies with the same relational core, yet still have different query mappings under their native semantic backgrounds. The difference in behavior of e_1 and e_2 cannot be determined from their syntax alone; it arises from their different native semantic conventions C_{SQL} and $C_{\mathrm{Soufflé}}$. If both expressions are instead interpreted under the same compatible conventions, say $C_{\mathrm{Soufflé}}$, then their query mappings agree:

$$[\![e_1]\!]_{\mathcal{S}_1,C_{\mathrm{Soufflé}}} = [\![e_2]\!]_{\mathcal{S}_1,C_{\mathrm{Soufflé}}}.$$

Both then return 0 for c on I_1. The difference under native interpretation therefore comes from the different native semantic conventions, not from the relational pattern structure.

The same relational structure in different notations. We call the common relational core underlying these conceptual evaluation strategies the *relational pattern structure* of an expression: an abstraction of how the expression derives its result from table-reference roles and relational constructs, independent of surface notation and semantic conventions. Thus, e_1 and e_2 instantiate the same relational pattern structure despite their different native semantic conventions.

The next example illustrates the distinction between relation names and the roles played by their occurrences (which dissociation later makes explicit). It also illustrates two further points: (*i*) equal query mappings need not imply equal pattern structures, and (*ii*) the same abstract pattern structure may occur over different schemas.

A repeated relation name. Consider a database schema $\mathcal{S}_2$ with the single relation schema R(A, B), primary key {A, B}, and NOT NULL constraints on both attributes. Figure 3a shows an admissible database instance I_2 over $\mathcal{S}_2$.

Suppose we want, for every value of A, the sum of the corresponding values of B. The SQL expression e_{group} in Fig. 3b states this directly with a GROUP BY clause. Soufflé has no GROUP BY clause. Thus, to express the same grouping task in Soufflé, e_3 in Fig. 3d refers to R twice: the outer atom R(x,_) supplies the A values that determine the groups, while R(x,y) supplies the matching B values to be summed.

Under their native conventions and over the admissible instances of $\mathcal{S}_2$, e_{group} and e_3 have the same query mapping (which we abbreviate as q_{group} and q_3) but different relational pattern structures. Expression e_{group} uses one table reference (an occurrence of a relation name in an input or binding position [30]) together with a grouping operator, whereas e_3 uses two correlated table references: the inner aggregate depends on the outer value x. Notice that these integrity constraints are crucial for this equality of query

R

A	B
a	1
a	2
b	3

(a) $\mathcal{S}_2, I_2$

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

(b) e_{group} **(SQL)**

Out

A	sm
a	3
b	3

(c) $q_{group}(I_2) = q_3(I_2)$

```
Q(x, sum y: {R(x,y)}) :- R(x,_).
```

(d) e_3 **(Soufflé)**

```
select RO.A,
 (select sum(RI.B) as sm
  from R RI
  where RI.A = RO.A)
from R RO
```

(e) e_4 **(SQL)**

Out

A	sm
a	3
a	3
b	3

(f) $q_4(I_2)$

Figure 3: A schema and instance (a), a grouped SQL expression and its result (b, c), a Soufflé expression with the same query mapping over $\mathcal{S}_2$ (d), and a correlated SQL expression with the same two-role pattern structure as e_3 but a duplicate-preserving result (e, f).

mappings: the key excludes duplicate base tuples that SQL would otherwise count with multiplicity, and the non-null constraints keep the inputs within Soufflé's value domain.

Expression e_4 in Fig. 3e is the SQL counterpart of e_3. Its aliases RI and RO name the inner and outer bindings, but both range over the same relation value I_2(R). The subquery is correlated because it refers to RO.A from the outer binding. Under native SQL bag semantics, each outer tuple produces a result tuple, so (a, 3) occurs twice. Under Soufflé's set-oriented conventions, the duplicate is collapsed. Thus, e_3 and e_4 have the same relational pattern structure but different native query mappings.

Relational pattern structure. Each table reference plays a distinct role in how the query conceptually derives its result. Thus, the two occurrences of R in e_3 and e_4 serve as separate inner and outer *table-reference roles*, even though ordinary query evaluation supplies both with the same relation value I_2(R). The roles and their relationships form part of the structural pattern of a query; dissociation will later make these roles explicit by assigning each a distinct relation name.

We write P_e for the *relational pattern structure* of expression e: its abstract relational structure, independent of surface notation and semantic conventions and, up to systematic renaming, of concrete schema names.[4] For the four expressions, $P_{e_1} = P_{e_2} = P_{e_3} = P_{e_4}$. We call this common structure CORRELATED SUBCOLLECTION SUMMARY:

> *Given an outer collection and an inner collection, use each outer tuple to determine a matching subcollection of inner tuples. Aggregate that subcollection, and output selected attributes of the outer tuple together with the resulting summary.*

The pattern has an outer role, an inner role, a matching condition, a copied outer value, and an aggregation step. It abstracts from surface

[4]The relational pattern structure abstracts from concrete schema vocabulary, while its *instantiations* are grounded in relation signatures, attributes, and compatible constructs. Textual representations of an abstract pattern typically still require names or explicit placeholders, while Relational Diagrams can identify table and attribute roles by graphical position and connections (without names or explicit placeholder tokens).

```
Q(x, sum y: {R1(x,y)}) :- R2(x,_).
```

(a) e'_3 (Soufflé)

```
select R0.A,
 (select sum(RI.B) as sm
  from R1 RI
  where RI.A = R0.A)
from R2 R0
```

(b) e'_4 (SQL)

Figure 4: Dissociation assigns distinct relation names to the inner and outer table-reference roles.

η	information need	information sought (often informal)
e	query expression	syntactic representation of the query
q	query mapping	function denoted by e
$q(I)$	query result	result of evaluating q on instance I
P_e	relational pattern structure	abstract relational structure of e
e'	dissociated query expression	e with distinct relation names for each table-reference role
q'	relational pattern denotation	semantics of P_e: query mapping of e' on independent role inputs

Figure 5: Overview of the main vocabulary and notation.

notation, systematic renaming, the particular aggregate function, and semantic conventions such as empty-aggregate behavior.

The query expression e_{group} is deliberately not among these four: it uses one table-reference role and an explicit grouping operator. Although e_{group} and e_3 have the same query mapping over $\mathcal{S}_2$, their pattern structures differ. This equality relies on the integrity constraints of $\mathcal{S}_2$; once they are removed, the structural difference can also become observable through different results on suitable inputs. We return to this point later.

Dissociation and relational pattern denotation. In ordinary evaluation of e_3 or e_4, both roles receive the same relation value $I_2(\mathsf{R})$. *Dissociation* replaces the relation name of each table reference with a distinct schema-level name, allowing the role inputs to vary independently. For e_3 and e_4, this yields the dissociated query expressions in Fig. 4. Notice the two new relation names R1 and R2, which may receive independent relation values. By contrast, the SQL aliases RI and RO only name local bindings and do not create independent schema inputs.

We write a database schema $\mathcal{S}$ as a pair $(\mathcal{R}_\mathcal{S}, \Gamma_\mathcal{S})$, where the relation-signature map $\mathcal{R}_\mathcal{S}$ maps relation names to relation signatures, and $\Gamma_\mathcal{S}$ is the set of integrity constraints. In this notation, $\mathcal{S}_2 = (\{\mathsf{R} : \tau_\mathsf{R}\}, \Gamma_{\mathcal{S}_2})$, with relation signature $\tau_\mathsf{R} = \{\mathsf{A} : \mathrm{string}, \mathsf{B} : \mathrm{int}\}$ and $\Gamma_{\mathcal{S}_2}$ containing the key and NOT NULL constraints.

The two dissociated expressions have the same constraint-free dissociated schema:

$$\mathcal{S}'_{e_3} = \mathcal{S}'_{e_4} = (\{\mathsf{R1} : \tau_\mathsf{R}, \mathsf{R2} : \tau_\mathsf{R}\}, \varnothing).$$

Both fresh names R1 and R2 inherit the signature of R, and $\varnothing$ records that none of the original integrity constraints is retained. A dissociated instance may assign them independent relation values (shown here with V_{inner} and V_{outer}):

$$I' = \{\mathsf{R1} \mapsto V_{\text{inner}},\ \mathsf{R2} \mapsto V_{\text{outer}}\}.$$

For $B = (\mathcal{S}, C)$, let $B'_e = (\mathcal{S}'_e, C)$. The *relational pattern denotation* is the query mapping of the dissociated expression:

$$q'_{e,B} \coloneqq [\![e']\!]_{B'_e}.$$

Notice that the two notions provide complementary views of relational patterns: P_e describes the pattern's table-reference roles and their structural relationships independently of C, while $q'_{e,B}$ gives its query mapping over independent role inputs under a fixed C and without the original integrity constraints.

The ordinary query result on I_2 is recovered by assigning (substituting) the original relation value to both roles:

$$I'_2 = \{\mathsf{R1} \mapsto I_2(\mathsf{R}),\ \mathsf{R2} \mapsto I_2(\mathsf{R})\}.$$

Then, for $e \in \{e_3, e_4\}$ and any fixed compatible C,

$$q_{e,(\mathcal{S}_2,C)}(I_2) = q'_{e,(\mathcal{S}_2,C)}(I'_2).$$

Thus, dissociation preserves ordinary query evaluation when corresponding roles receive the same value, while also allowing their values to vary independently.

Pattern isomorphism across schemas and languages. Dissociation exposes the inner and outer roles of all four expressions as independently named inputs. A *pattern-preserving schema mapping* λ aligns the roles and query-relevant attributes of the R expressions with those of the employee-and-sales expressions:

$$\mathsf{R1}(\mathsf{A}, \mathsf{B}) \leftrightarrow \mathsf{Sales}(\mathsf{eid}, \mathsf{val}), \qquad \mathsf{R2}(\mathsf{A}) \leftrightarrow \mathsf{Empl}(\mathsf{eid}),$$

together with A $\leftrightarrow$ eid and sm $\leftrightarrow$ sm in the output. The unused attributes Sales.date and Empl.k do not participate in the alignment.

Informally, two expressions are *pattern-isomorphic* if such a bijective alignment makes their dissociated denotations equal on every aligned dissociated input under every shared compatible choice of semantic conventions C. The alignment above therefore shows that e_1, e_2, e_3, and e_4 are pairwise pattern-isomorphic: they instantiate the same correlated subcollection summary over different schemas and notations.

The examples distinguish three levels at which queries may be called "the same": they may (*i*) address the same *information need* η, (*ii*) have the same *query mapping* $q_{e,B}$ under a semantic background B, or (*iii*) have pattern-isomorphic *relational pattern structures*. Expressions e_1 and e_2 share the first and third properties but have different query mappings under their native conventions. Expressions e_{group} and e_3 have the same query mapping over $\mathcal{S}_2$ but different pattern structures. Finally, $e_1, \ldots, e_4$ are pairwise pattern-isomorphic.

Keeping these concepts distinct is essential: without the notion of relational pattern, debates on language design tend to focus on logical expressiveness [29, 30]. And without clearly separating semantic conventions from syntax, comparisons may attribute to notational choices effects that are actually due to background context not directly encoded in a query expression.

In our framework, a query expression can be seen as a derived notion: it encodes (or instantiates) a relational pattern structure in a notation; under a fixed semantic background, it denotes a query mapping.

	e_1	e_2	e_4	e_3	e_{group}
Informal information need η	Employee sales total		Sum of B for each value of A		
Query mapping $q_{e,B}$	$q_{e_1,(\mathcal{S}_1,C_{\text{SQL}})}$	$q_{e_2,(\mathcal{S}_1,C_{\text{Soufflé}})}$	$q_{e_4,(\mathcal{S}_2,C_{\text{SQL}})}$	$q_{e_3,(\mathcal{S}_2,C_{\text{Soufflé}})} = q_{e_{group},(\mathcal{S}_2,C_{\text{SQL}})}$	
Relational pattern structure P_e	Correlated subcollection summary				Grouped aggregate

Figure 6: Relationships among the five query expressions at three levels. Within each row, a contiguous shaded block groups expressions that share the same information need, query mapping, or relational pattern structure, respectively (gray shades have no meaning across rows). Query mappings use each expression's native semantic conventions. The equality of the query mapping between e_3 and e_{group} is relative to the admissible instances of $\mathcal{S}_2$, including its key and non-null constraints.

3 LEARNING OUTCOMES, AUDIENCE, SCOPE

Attendees will leave with a sharper vocabulary for discussing relational query languages and more informed mental models of the design trade-offs between different notations. They will learn to recognize recurring relational patterns across different notations. Importantly, these patterns exist independently of any particular notation. Rather than reproducing examples chosen by recent language proposals, the tutorial starts from a common set of representative queries and examines how different languages express this shared workload. Along the way, it introduces well-established concepts in relational language design (such as domain vs. tuple perspective, and named vs. positional access to relation components) and more recent terminology from the author's work (such as 'inside-out' vs. 'outside-in'). These concepts are illustrated first through running examples and only then formalized. Thus instead of starting from abstract concepts, we develop them by looking at many examples. By the end of the tutorial, attendees should be able to look at an unfamiliar relational language, separate relational pattern from semantic conventions and notation, and explain the main design trade-offs that follow.

Audience and prerequisites. This 3-hour tutorial targets researchers and practitioners seeking an intuitive guide to relational language design. It focuses on the commonalities and differences across various languages. Familiarity with SQL is expected. Basic knowledge of relational algebra (RA) and relational calculus (RC) is helpful but not required.

Distribution. Slides will be made available after the tutorial on the tutorial webpage, as with other recent tutorials by the presenter and collaborators [25, 26, 69, 70].

Relation to prior tutorials. This 3-hour tutorial extends a prior 90-minute tutorial from SIGMOD 2026 [28]. The extensions are described in the outline further below.

Recent tutorials [10, 22, 49, 53] and surveys [6] on graph query languages focus on graph data models and graph query languages in their own right. Our use of graph and path notation is instead limited to a focused comparison within a broader tutorial on *relational* language design. It remains to be seen which aspects of graph query languages are currently missing in relational languages, and whether the separation into these two different camps is really necessary. This raises a broader question: is the divide between relational and graph languages partly artificial, and can a more systematic view of relational language design help dissolve it?

4 TUTORIAL CONTENT AND OUTLINE

The tutorial is organized around a small set of representative SQL queries that are translated into multiple relational languages and shown side by side. Rather than defining terminology up front, we introduce it through comparisons of concrete query expressions (e.g., contrasting domain and tuple relational calculus) and only then distill the recurring structural notation choices. This example-first organization matches the overall goal of the tutorial: to separate query mapping, relational pattern, structural notation, and semantic conventions, and to give attendees a vocabulary for recognizing the same relational structure across notationally different languages.

Core concepts. Using ARC as a common reference representation and Relational Diagrams as an auxiliary visual aid, we introduce the main concepts that recur across relational languages. These include binding structures (tuple vs. domain variables), named vs. positional access to relation components, pointwise (variable-based) vs. point-free (tacit) notation [7, 11, 12], set vs. bag semantics (and list semantics, where relevant), nested vs. pipelined composition, declarative vs. procedural style [73], null handling and binary vs. ternary logic [50], explicit joins via equality predicates, implicit joins via shared variables, joins via path expressions [24, 75], grouping from the inside out (FIO) vs. from the outside in (FOI) [32], negation patterns, correlated subqueries, scalar subqueries, lateral joins, and derived attributes. We relate these choices to different user tasks: composing and reading queries [62], and revising existing queries. We also borrow concepts from programming-language design, especially orthogonality [12, 46], and will discuss abstraction (the art of omitting unnecessary detail).

Languages. We begin with the classical core languages SQL [38], tuple relational calculus (TRC) [27], domain relational calculus (DRC), relational algebra (RA), and Datalog [18]/Soufflé [2, 63], to establish the main concepts. We discuss SQL's conceptual evaluation strategy, the connection to nested-loop or nested-generators in set comprehension [36], and illustrate comprehension-based code in both Haskell [45, 72] and Python [71]. We examine functional-style languages [35, 36, 58] (such as DAPLEX [65]) and Boute's formalism [12]. We discuss Rel [7] and Morel [42] as two recent relational programming languages that revisit the boundary between query language and host languages, and extend relational querying toward programming in the large. We take a deeper look at formalisms for aggregation, starting from Klug's formalism for aggregate queries under set semantics [48], which influenced subsequent comprehension-based models for database programming languages (DBPLs) [15, 23, 37, 68], including extensions of logic

with aggregate operators [40]. Finally, we discuss recent compositional proposals such as Google's pipe syntax [66], SaneQL [57], and PRQL [1], asking how pipelining affects modularity and ease of reading queries. Depending on time, we briefly connect these ideas to dataframe-style systems such as pandas [54] and Ibis [55] and to point-free array languages such as APL [43] and J [44].

Extensions over SIGMOD'26 tutorial. We introduce *recursion* through Datalog and recursive SQL, covering alternatives to conventional fixpoint evaluation [41] and recursion with aggregation [47]. We compare path-query notation across relational, graph, and deductive languages [64], including GQL and SQL/PGQ graph patterns [21] and GEM's *functional joins* [75].

We look beyond flat relations to nested relational data. Building on general models and languages for complex values [3], we compare SQL++ [17] with nested-value and array support in SQL. The examples expose differences in nesting and unnesting, grouping, and order (unordered collections versus ordered arrays or lists).

Finally, we survey three ways to declaratively express NP search problems and problems at higher levels of the polynomial hierarchy: (*i*) normal logic programs with recursion and default negation, interpreted under stable-model semantics, a core fragment of answer-set programming (ASP) [13]; (*ii*) existentially quantified relation variables in existential second-order logic (ESO), which captures NP over finite structures, together with database-language realizations [16]; and (*iii*) disjunctive rule heads, yielding disjunctive logic programs [20]. The first two support NP-level search, whereas the third reaches the second level of the polynomial hierarchy.

Stable-model semantics has been used in databases, for example, for consistent query answering over inconsistent databases [9] and trust-based conflict resolution [33]. Related database languages and systems expose combinatorial choice and optimization through different mechanisms, including free second-order relation variables in LogicBlox [8], solver-integrated SQL in SolveDB [67], package queries [14], the DeciDB prototype [51], and polymorphic SQL [60]. If time permits, we discuss how the ASP system clingo [34, 59] permits disjunctive ASP that can automate hardness reductions [52].

5 AUTHOR INFORMATION

Wolfgang Gatterbauer is an Associate Professor at the Khoury College of Computer Sciences at Northeastern University. His research interests lie at the intersection of theory and practice of data management. He has given tutorials on visual query representations at VLDB'23 [25] and ICDE'24 [26] and on query optimization at SIGMOD'20 [69] and ICDE'22 [70]. Slides from those tutorials are available on the tutorial pages, including recorded videos for one of them. The content of this tutorial is heavily informed by his work on visual representations of relational queries [27, 29–31], by the recent proposal of an abstract relational query language that can embed other relational languages and represent their patterns [32], and by a prior shorter tutorial on the same topic at SIGMOD 2026 [28].

ACKNOWLEDGMENTS

Thanks to Molham Aref, Diandre Sabale, and Val Tannen for helpful comments. A large language model (LLM) was used to refine the wording and improve the clarity of this tutorial proposal.

REFERENCES

[1] 2025. PRQL (Pipelined Relational Query Language). https://prql-lang.org/
[2] 2025. Soufflé. Aggregates and Generative Functors. https://souffle-lang.github.io/aggregates.
[3] Serge Abiteboul and Catriel Beeri. 1995. The Power of Languages for the Manipulation of Complex Values. *VLDB J.* 4, 4 (1995), 727–794. http://www.vldb.org/journal/VLDBJ4/P727.pdf
[4] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/
[5] Anastasia Ailamaki, Samuel Madden, Daniel Abadi, Gustavo Alonso, Sihem Amer-Yahia, Magdalena Balazinska, Philip A. Bernstein, Peter 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
[6] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. doi:10.1145/3104031
[7] 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
[8] Molham Aref, Benny Kimelfeld, Emir Pasalic, and Nikolaos Vasiloglou. 2015. Extending Datalog with Analytics in LogicBlox. In *Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management*, Vol. 1378. https://ceur-ws.org/Vol-1378/AMW_2015_paper_3.pdf
[9] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. 2003. Answer sets for consistent query answering in inconsistent databases. *Theory Pract. Log. Program.* 3, 4-5 (2003), 393–424. doi:10.1017/S1471068403001832
[10] Marcelo Arenas, Claudio Gutierrez, and Juan F. Sequeda. 2021. Querying in the Age of Graph Databases and Knowledge Graphs. In *SIGMOD*. 2821–2828. doi:10.1145/3448016.3457545
[11] John W. Backus. 1978. Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs. *CACM* 21, 8 (1978), 613–641. doi:10.1145/359576.359579
[12] Raymond T. Boute. 2005. Functional declarative language design and predicate calculus: a practical approach. *ACM Trans. Program. Lang. Syst.* 27, 5 (2005), 988–1047. doi:10.1145/1086642.1086647
[13] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. 2011. Answer set programming at a glance. *Commun. ACM* 54, 12 (2011), 92–103. doi:10.1145/2043174.2043195
[14] Matteo Brucato, Azza Abouzied, and Alexandra Meliou. 2018. Package queries: efficient and scalable computation of high-order constraints. *VLDB J.* 27, 5 (2018), 693–718. doi:10.1007/S00778-017-0483-4
[15] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. 1994. Comprehension Syntax. *SIGMOD Rec.* 23, 1 (1994), 87–96. doi:10.1145/181550.181564
[16] Marco Cadoli and Toni Mancini. 2007. Combining relational algebra, SQL, constraint modelling, and local search. *Theory Pract. Log. Program.* 7, 1-2 (2007), 37–65. doi:10.1017/S1471068406002857
[17] 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
[18] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (1989), 146–166. doi:10.1109/69.43410
[19] E. F. Codd. 1972. Relational Completeness of Data Base Sublanguages. *Research Report / RJ / IBM / San Jose, California* RJ987 (1972). https://citeseerx.ist.psu.edu/document?doi=6a048dc38250ffce49c5e6a5040b4c91ca05e83d
[20] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33, 3 (2001), 374–425. doi:10.1145/502807.502810
[21] Alin Deutsch, Nadime Francis, Alastair Green, Keith W. Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD*. ACM, 2246–2258. doi:10.1145/3514221.3526057
[22] Alin Deutsch and Yannis Papakonstantinou. 2018. Graph data models, query languages and programming paradigms. *PVLDB* 11, 12 (Aug. 2018), 2106–2109. doi:10.14778/3229863.3229879

[23] Leonidas Fegaras and David Maier. 2000. Optimizing object queries using an effective calculus. *ACM TODS* 25, 4 (2000), 457–516. doi:10.1145/377674.377676

[24] Jürgen Frohn, Georg Lausen, and Heinz Uphoff. 1994. Access to Objects by Path Expressions and Rules. In *VLDB*. 273–284. http://www.vldb.org/conf/1994/P273.PDF

[25] Wolfgang Gatterbauer. 2023. A Tutorial on Visual Representations of Relational Queries. *PVLDB* 16, 12 (2023), 3890–3893. doi:10.14778/3611540.3611578 Tutorial page with slides: https://northeastern-datalab.github.io/visual-query-representation-tutorial/.

[26] 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 Tutorial page with slides: https://northeastern-datalab.github.io/diagrammatic-representation-tutorial/.

[27] Wolfgang Gatterbauer. 2026. A Principled Solution to the Disjunction Problem of Diagrammatic Query Representations. *PACMMOD* 4, 1 (2026), 3:1–3:28. doi:10.1145/3786617 Full version: https://arxiv.org/pdf/2412.08583.

[28] Wolfgang Gatterbauer. 2026. A Tutorial on Relational Language Design. In *SIGMOD tutorials*. 550–555. doi:10.1145/3788853.3801875

[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 Full version: https://arxiv.org/pdf/2401.04758.

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

[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] Wolfgang Gatterbauer and Diandre Miguel Sabale. 2026. Database Research Needs an Abstract Relational Query Language. In *CIDR*. https://vldb.org/cidrdb/papers/2026/p27-gatterbauer.pdf Project webpage with slides and video: https://relationaldiagrams.com/.

[33] Wolfgang Gatterbauer and Dan Suciu. 2010. Data conflict resolution using trust mappings. In *SIGMOD*. ACM, 219–230. doi:10.1145/1807167.1807193

[34] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19, 1 (2019), 27–82. doi:10.1017/S1471068418000054

[35] Peter M. D. Gray. 2009. Functional Query Language. In *Encyclopedia of Database Systems*. Springer US, 1201–1204. doi:10.1007/978-0-387-39940-9_1092

[36] Peter M. D. Gray, Peter J. H. King, and Alexandra Poulovassilis. 2004. *Introduction to the Use of Functions in the Management of Data*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–54. doi:10.1007/978-3-662-05372-0_1

[37] 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

[38] Paolo Guagliardo and Leonid Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *PVLDB* 11, 1 (2017), 27–39. doi:10.14778/3151113.3151116

[39] Paul R. Halmos. 1985. *I Want to be a Mathematician: An Automathography*. Springer. doi:10.1007/978-1-4612-1084-9

[40] 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

[41] Denis Hirn and Torsten Grust. 2023. A Fix for the Fixation on Fixpoints. In *CIDR*. https://vldb.org/cidrdb/2023/a-fix-for-the-fixation-on-fixpoints.html

[42] Julian Hyde. 2026. Morel. https://github.com/hydromatic/morel

[43] Kenneth E. Iverson. 1980. Notation as a tool of thought. *CACM* 23, 8 (Aug. 1980), 444–465. doi:10.1145/358896.358899

[44] Kenneth E. Iverson and Roger Hui. 2026. J. https://www.jsoftware.com/

[45] Simon L. Peyton Jones and Philip Wadler. 2007. Comprehensive comprehensions. In *Proc. ACM SIGPLAN Workshop on Haskell, (Haskell 2007)*. ACM, 61–72. doi:10.1145/1291201.1291209

[46] Guy L. Steele Jr. 1999. Growing a Language. *High. Order Symb. Comput.* 12, 3 (1999), 221–236. doi:10.1023/A:1010085415024

[47] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. Datalog in Wonderland. *SIGMOD Rec.* 51, 2 (2022), 6–17. doi:10.1145/3552490.3552492

[48] 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

[49] Haridimos Kondylakis, Stefania Dumbrava, Matteo Lissandrini, Nikolay Yakovets, Angela Bonifati, Vasilis Efthymiou, George Fletcher, Dimitris Plexousakis, Riccardo Tommasini, Georgia Troullinou, and Elisjana Ymeralli. 2025. Property Graph Standards: State of the Art & Open Challenges. *PVLDB* 18, 12 (2025), 5477–5481. doi:10.14778/3750601.3750698

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

[51] Anh L. Mai, Filip Milisov, Hatim Rehmanjee, Azza Abouzied, Peter J. Haas, and Alexandra Meliou. 2026. DeciDB: In-Database Processing of Decision Queries. (2026). https://decidb.com/about.html (unpublished pre-print).

[52] Neha Makhija and Wolfgang Gatterbauer. 2023. A Unified Approach for Resilience and Causal Responsibility with Integer Linear Programming (ILP) and LP Relaxations. *PACMMOD* 1, 4 (2023), 228:1–228:27. doi:10.1145/3626715

[53] Ioana Manolescu and Madhulika Mohanty. 2023. Full-Power Graph Querying: State of the Art and Challenges. *PVLDB* 16, 12 (Aug. 2023), 3886–3889. doi:10.14778/3611540.3611577

[54] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proc. 9th Python in Science Conference*. 56–61. doi:10.25080/Majora-92bf1922-00a

[55] Wes McKinney and Ibis Project authors. 2026. The Ibis Project: the portable Python dataframe library. Version 12.0.0. 2026. https://ibis-project.org/

[56] Richard Morris. 2013. Don Knuth and the Art of Computer Programming: The Interview. https://www.red-gate.com/simple-talk/opinion/opinion-pieces/don-knuth-and-the-art-of-computer-programming-the-interview/

[57] 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

[58] Norman W. Paton and Peter M. D. Gray. 1990. Optimising and Executing DAPLEX Queries Using Prolog. *Comput. J.* 33, 6 (1990), 547–555. doi:10.1093/COMJNL/33.6.547

[59] Potassco: the Potsdam Answer Set Solving Collection. 2022. clingo. https://potassco.org/clingo/

[60] David Robert Pratten, Luke Mathieson, and Fahimeh Ramezani. 2025. Relational Algebras for Subset Selection and Optimisation. arXiv:2509.06439 [cs.DB] https://arxiv.org/abs/2509.06439

[61] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems* (3 ed.). McGraw-Hill, Inc., USA. https://dl.acm.org/doi/book/10.5555/560733

[62] Phyllis Reisner. 1981. Human Factors Studies of Database Query Languages: A Survey and Assessment. *ACM Comput. Surv.* 13, 1 (1981), 13–31. doi:10.1145/356835.356837

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

[64] Amir Shaikhha, Youning Xia, Meisam Tarabkhah, Jazal Saleem, and Anna Herlihy. 2026. Raqlet: Cross-Paradigm Compilation for Recursive Queries. In *CIDR*. https://vldb.org/cidrdb/2026/raqlet-cross-paradigm-compilation-for-recursive-queries.html

[65] David W. Shipman. 1981. The functional data model and the data languages DAPLEX. *ACM TODS* 6, 1 (March 1981), 140–173. doi:10.1145/319540.319561

[66] 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

[67] Laurynas Siksnys and Torben Bach Pedersen. 2016. SolveDB: Integrating Optimization Problem Solvers Into SQL Databases. In *SSDBM*. ACM, 14:1–14:12. doi:10.1145/2949689.2949693

[68] Philip W. Trinder. 1991. Comprehensions, a Query Notation for DBPLs. In *3rd international workshop on Database Programming Languages (DBPL)*. 55–68. https://dl.acm.org/doi/10.5555/135260.135271

[69] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. In *SIGMOD tutorials*. ACM, 2659–2665. doi:10.1145/3318464.3383132 Tutorial page with slides: https://northeastern-datalab.github.io/topk-join-tutorial/.

[70] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2022. Toward Responsive DBMS: Optimal Join Algorithms, Enumeration, Factorization, Ranking, and Dynamic Programming. In *ICDE tutorials*. IEEE, 3205–3208. doi:10.1109/ICDE53745.2022.00299 Tutorial page with slides and videos: https://northeastern-datalab.github.io/responsive-dbms-tutorial/.

[71] Guido van Rossum. 2026. Python. http://www.python.org

[72] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, Gilles Kahn (Ed.). ACM, 61–78. doi:10.1145/91556.91592

[73] Charles Welty and David W. Stemple. 1981. Human Factors Comparison of a Procedural and a Nonprocedural Query Language. *ACM TODS* 6, 4 (1981), 626–649. doi:10.1145/319628.319656

[74] Wikipedia. 2026. Linguistic relativity. https://en.wikipedia.org/wiki/Linguistic_relativity [Online; accessed March-2026].

[75] Carlo Zaniolo. 1983. The database language GEM. *SIGMOD Rec.* 13, 4 (May 1983), 207–218. doi:10.1145/971695.582226