Relational Diagrams and the Pattern Expressiveness of Relational Languages

Wolfgang Gatterbauer
Northeastern University
Boston, MA, USA

Cody Dunne
Northeastern University
Boston, MA, USA

ABSTRACT

Comparing relational languages by their logical expressiveness is well understood. Less understood is how to compare relational languages by their ability to represent *relational query patterns*. Indeed, what are query patterns other than "a certain way of writing a query"? And how can query patterns be defined across procedural and declarative languages, irrespective of their syntax?

Our SIGMOD 2024 paper proposes a semantic definition of relational query patterns that uses a variant of structure-preserving mappings between the relational tables of queries. This formalism allows us to analyze the *relative pattern expressiveness* of relational languages. Notably, for the non-disjunctive language fragment, we show that relational calculus (RC) can express a larger class of patterns than the basic operators of relational algebra (RA).

We also propose Relational Diagrams, a complete and sound diagrammatic representation of safe relational calculus. These diagrams can represent all query patterns for unions of non-disjunctive queries, in contrast to visual query representations that derive visual marks from the basic operators of algebra. Our anonymously preregistered user study shows that Relational Diagrams allow users to recognize relational patterns meaningfully faster and more accurately than they can with SQL.

1 Introduction

When designing and comparing query languages, we are usually concerned with *logical expressiveness*: can a language express a particular query or not? For relational languages, questions of expressiveness have been studied at least since Codd proposed that first-order predicate calculus could serve as a standard for measuring the expressive power of query languages. Today, formalisms for comparing expressiveness are well-developed and understood. We do not yet have a similarly developed machinery to reason about *relational query patterns*. Intuitively, a query pattern should capture "a certain way of writing a query" and be applicable across all major relational languages.

What is a pattern and why do we care? A pattern is defined as "something used as a model for making things" [17]. In software engineering, a design pattern is a *general and reusable solution* to a common coding problem. The goal of studying patterns is to abstract *templates* for solving particular types of problems that can be *reused* across different situations and *speed up* the development process [31]. These templates or motifs typically describe ways in which a set of program constituents (e.g., classes or objects) relate to each other in order to achieve a design goal [31]. The core ideas are usually abstracted away from a particular programming language. Similarly, we aim to formalize *the different ways in which the constituents of relational queries (the tables) relate to each other in order to describe query intent*, across different procedural and logical relational languages.

We posit that identifying patterns in queries could open novel paths for assisting users [14], especially learners trying to understand the structure behind relational queries written in different languages. Relational patterns could help learners spot similar queries across different schemas, and thus more easily separate the logic (or motif) from the syntactic expression. Learners could compare queries based on their logic rather than language-specific syntax.

Relational patterns. We suggest a semantic definition of relational patterns that allows us to compare the patterns of queries across different schemas and query languages. In other words, two logically equivalent queries may use different patterns; and two queries over different schemas may use the same pattern. An important insight that we established in [12] is that *the fundamental operators of Relational Algebra (RA) are insufficient to express all query patterns from Relational Calculus (RC)*. Instead, RA requires us to reformulate queries and repeat base tables more often. It follows that any diagrammatic query representation that maps RA operators onto visual marks is inherently unable to express the full range of relational query patterns from RC.

EXAMPLE 1 (Universal queries). *Consider query Q_1 "Find the names of sailors who have reserved all boats" from the cow textbook [21, Sect. 4.3.1]. It can be written in Tuple Relational Calculus (TRC) as follows (Q_1^1):*

$$\{Q_1^1(sname) \mid \exists s \in Sailor[Q_1^1.sname = s.sname \wedge \forall b \in Boat[\exists r \in Reserves[s.sid = r.sid \wedge r.bid = b.bid]]]\} \quad (1)$$

Figure 1a shows the query in SQL as Q_1^2. It can also be written in Relational Algebra (RA) extended with antijoin operator $\triangleright$ as (2) and Datalog with negation as (3):

$$Q_1^3 = \pi_{sname}(Sailor \triangleright \pi_{sid}((\pi_{sid}Sailor \times \pi_{bid}Boat) - \pi_{sid,bid}Reserves)) \quad (2)$$

This is a summary and minor revision of the paper titled "On The Reasonable Effectiveness of Relational Diagrams: Explaining Relational Query Patterns and the Pattern Expressiveness of Relational Languages," published in PACMMOD, Vol. 2, No. 1 (SIGMOD), Article 61, February 2024. DOI: https://doi.org/10.1145/3639316.

```sql
SELECT DISTINCT S.sname
FROM Sailor S
WHERE NOT EXISTS (
  SELECT *
  FROM Boat B
  WHERE NOT EXISTS (
    SELECT *
    FROM Reserves R
    WHERE R.bid = B.bid
    AND R.sid = S.sid))
```

(a) Q_1^2 in Example 1

```sql
SELECT DISTINCT S1.sname sname
FROM Sailor1 S1
WHERE not EXISTS(
  SELECT *
  FROM Boat B, Sailor2 S2
  WHERE S2.sid = S1.sid
  AND not EXISTS(
    SELECT *
    FROM Reserves R
    WHERE R.bid = B.bid
    AND R.sid = S2.side))
```

(b) Q_9^{1*} in Example 11

```sql
SELECT DISTINCT T1.att3 att1
FROM Table1 T1
WHERE NOT EXISTS(
  SELECT *
  FROM Table2 T2, Table4 T4
  WHERE T4.att7 = T1.att2
  AND NOT EXISTS(
    SELECT *
    FROM Table3 T3
    WHERE T3.att6 = T2.att4
    AND T3.att5 = T4.att7))
```

(c) Q_9^1 in Example 11

Figure 1: Universal SQL queries used in Example 1 and Example 11.

$$\begin{aligned} I(s,b) &:\!- \mathit{Reserves}(s,b,\_).\\ J(s) &:\!- \mathit{Sailor}(s,\_,\_,\_), \mathit{Boat}(b,\_,\_,), \neg I(s,b).\\ Q_1^4(n) &:\!- \mathit{Sailor}(s,n,\_,\_), \neg J(s). \end{aligned} \tag{3}$$

Notice that the above RA and Datalog expressions for Q_1 use the Sailor relation twice. The appendix to our SIGMOD 2024 paper shows that Q_1 cannot be represented in RA or Datalog using each of the 3 base tables only once and without a dedicated relational division operator.

Next, consider the query Q_2 "Get supplier names for suppliers who supply all parts" from Date's book [5, Sect. 8.3], written in his TRC notation as Q_2^1:

```
SX.SNAME WHERE NOT EXISTS PX (NOT EXISTS SPX
  (SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO))
```
(4)

From the natural language description, both queries Q_1 and Q_2 seem to express a similar intent. However, the similarity is hard to see from the query expressions Q_1^1 (1) and Q_2^1 (4), since they use different schemas and quite different notations for TRC. Our semantic definition of relational query patterns allows us to formally state that queries Q_1^1 (1) and Q_2^1 (4) in TRC use the same relational pattern, also Q_1^3 (2) in RA and Q_1^4 (3) in Datalog use the same relational pattern, while Q_1^1 (1) in TRC and Q_1^3 (2) in RA use different relational patterns.

Our 1st contribution: relational patterns. We develop a language-independent notion of relational query patterns that allows us to decide whether two queries use the same pattern. Our definition is semantic—in the sense that the definition involves relations over sets of attributes—instead of syntactic—which would involve structural properties that are inherently language-dependent. Intuitively, our formalism reasons about mappings between the (existentially or universally) quantified relations referenced in two queries. We show how to use this formalism to compare relational query languages by their ability to express query patterns present in other languages and thus compare their relative pattern expressiveness. In particular, we contribute a novel hierarchy of pattern expressiveness among the non-disjunctive fragments of four relational query languages.

Our 2nd contribution: Relational Diagrams. We formalize diagrammatic representations of relational queries that capture the semantic relationships between its table references. We prove they are unambiguous (every diagram has a unique logical interpretation), they are relationally complete (every relational query can be expressed in a Relational Diagram), and they can express all patterns in the relational query language fragment of unions of non-disjunctive queries.

Our paper [12] and its supplemental material [13] includes many illustrated examples, proofs, and a benchmark of relational queries from 5 popular database textbooks. It also reports on a preregistered user study with a reproducible and scalable study design, showing that Relational Diagrams can help users recognize patterns faster and with higher accuracy than SQL. Two recent tutorials at VLDB 2023 [10] and ICDE 2024 [8] include an extensive discussion of related work. We also recommend an 18-min recorded video.[1]

2 Relational (Query) Patterns

Example 1 illustrated that—while two languages may have the same logical expressiveness—one of them may have more ways to represent logical patterns than the other. To make this intuition precise, we propose *a language-agnostic formalism* that defines relational (query) patterns and applies to any relational query language—regardless of its syntax. This formalism allows us to then study the relative pattern expressiveness of languages, i.e., *can language $\mathcal{L}_2$ express all patterns from language $\mathcal{L}_1$?*

We often need to distinguish between a query's *expression* (the actual syntax in a particular query language) and a query's *function* (mapping a set of base tables to an output table). We use uppercase Q for the *intent* of a query, subscript-indexed Q_i for different queries and superscript-indexed Q^j if we use alternative *expressions* for the same intent. Lowercase q^j denotes the *function* (or semantics) implied by expression Q^j.

Intuition. Our idea is to formalize patterns based on the only common constituents in queries across languages: references to the *base tables* of a database. Since every relational query language needs to reference tables, the resulting formalism generalizes. Intuitively, we define two queries to be *pattern-isomorphic*[2] if there is a one-to-one correspondence that pairs each table in one query with a table in the other query that "plays the same semantic role." This means that, when applying identical changes to these paired base tables (e.g. inserting a tuple), both queries will make identical changes to their outputs. However, for queries with multiple references to the same base table (also called self-joins), we need to treat such repeated "table references" as if they were referring to different and thus *independent tables*.

[1] SIGMOD talk: https://www.youtube.com/watch?v=IVRPD-074Ro

[2] Recall that an *isomorphism* is a reversible *structure-preserving* mapping between two structures. For it to be reversible, it needs to be surjective (each element in the target is mapped to) and injective (different elements in the source map to different elements in the target) [7]. We use the terms *pattern-isomorphic* and pattern-equivalent (instead of structure-isomorphic) since our focus is on particular relational structures we refer to as patterns.

For example, consider $Q = R - (\pi_A R \times T)$. The semantics of this query is a function $q(R, T)$ that maps base tables R and T to an output table Q. However, we are not interested in the argument list of a query's function, but rather the "signature" of a query's expression (i.e., the different references to base tables) since we want to capture that the two references to R play different semantic roles in the query. To capture these different roles, we assign *unique names to each table reference*, resulting in what we call the *dissociated query* $Q' = R_1 - (\pi_A R_2 \times T)$. We then formally define the *relational (query) pattern* of Q as the function $q'(R_1, R_2, T)$ expressed by the dissociated query Q'.[3]

2.1 Defining Relational Patterns

We assume that a query does not reuse views, i.e., each derived table—IDBs (Intensional Database predicates) in Datalog or CTEs (Common Table Expressions) in SQL—is referenced only once. (We will remove this assumption later in Section 2.3.) To apply our definition across different relational languages with varying syntax, we need a precise way to refer to each instance of a base table in a query expression.

Definition 2 (Query signature). *A* table reference *in a query expression Q is any reference of a base table as a part of a binding construct. The* signature $\mathcal{S}$ *of Q is the ordered sequence of table references as they appear in Q.*

For example, the symbol "R" is a table reference in the SQL fragment "`FROM R as R1`", the TRC fragment "$\exists r_1 \in R$", the RA fragment "$\pi_A R$", and the Datalog fragment "$R(x, \_)$".[4] In contrast, the symbol "R" is not a table reference in the SQL fragment "`WHERE R.A=1`" but part of an attribute reference using a previously-defined table variable "R" (called relational variable or relvar by Date and Darwen [4]). The signature of a conjunctive SQL query with FROM clause "`FROM R as R1, R as R2, T`" is $\mathcal{S} = (R, R, T)$ whereas the list of table variables is $(R1, R2, T)$.

Definition 3 (Dissociated query). *A dissociation of a query expression Q with signature $\mathcal{S}$ is a modified query Q', in which $\mathcal{S}$ is replaced with a signature $\mathcal{S}'$ of the same arity. Each table in $\mathcal{S}'$ has a distinct name but inherits the same schema as its corresponding table in $\mathcal{S}$.*

We call $\mathcal{S}'$ the dissociated signature of Q. It is easy to dissociate a query by simply replacing duplicate table references in $\mathcal{S}$ with fresh table names. For simplicity, we use subscripts when dissociating tables.

Example 4 (Dissociation). *The RA query*

$$Q_3 = R - (\pi_A R \times T) \tag{5}$$

has signature $\mathcal{S}_3 = (R, R, T)$ with two table references to the same base table R. Replacing $\mathcal{S}_3$ with a dissociated signature $\mathcal{S}'_3 = (R_1, R_2, T_3)$ gives the dissociated query $Q'_3 = R_1 - (\pi_A R_2 \times T_3)$. Since the dissociated tables inherit the schemas from their base tables, the dissociated query is still valid and represents a new function $q'_3(R_1, R_2, T_3)$ that maps three different base tables to an output.

The intuition is that the dissociated query defines a function that maps a list of *table references* (instead of an argument list of base tables) to an output table. Thus, *the dissociated query is a semantic definition of a relational query pattern* across different relational query languages.

Definition 5 (Relational pattern). *Given a query expression Q with signature $\mathcal{S}$, the* relational pattern *of Q is the function $q'(\mathcal{S}')$ defined by its dissociated query Q'.*

Two logically-equivalent queries use the same query pattern if their dissociated queries remain logically-equivalent after renaming of the base tables. We define this equivalence in terms of permutations applied to the arguments of their functions. For example, $f(x, y) = xy^2$ and $g(z, w) = z^2 w$ are equivalent under permutation $\pi = (2, 1)$, as $f(x, y) = g(\pi(x, y)) = g(y, x)$.

Definition 6 (Pattern isomorphism (preliminary)). *Two queries Q_a and Q_b with dissociated queries Q'_a and Q'_b and dissociated signatures $\mathcal{S}'_a$ and $\mathcal{S}'_b$, respectively, are* pattern-isomorphic *(i.e., "they have the same pattern") if, for some permutation π, the output of Q'_a on $\mathcal{S}'_a$ is equal to the output of Q'_b on $\pi(\mathcal{S}'_a)$, i.e., $q'_a(\mathcal{S}'_a) = q'_b(\pi(\mathcal{S}'_a))$. In that case, the bijection $\mathcal{S}_a[i] \mapsto \mathcal{S}_b[\pi(i)]$ between the table references is called a* pattern-preserving mapping *from Q_a to Q_b.*

Example 7 (Patterns). *Consider the Datalog query*

$$\begin{aligned} I(x, y) &:\!- R(x, \_), T(y). \\ Q_4(x, y) &:\!- R(x, z), \neg I(x, y). \end{aligned} \tag{6}$$

with signature $\mathcal{S}_4 = (R, T, R)$. A dissociated query is

$$\begin{aligned} I(x, y) &:\!- R_4(x, \_), T_5(y). \\ Q'_4(x, y) &:\!- R_6(x, z), \neg I(x, y). \end{aligned}$$

with signature $\mathcal{S}'_4 = (R_4, T_5, R_6)$ that defines a function $q'_4(R_4, T_5, R_6)$ mapping three base tables to a binary output.

Notice that the query Q_4 (6) in Datalog has the same pattern as Q_3 (5) in RA since their dissociated queries define the same function up to permutation of the arguments: $q'_3(R_1, R_2, T_3) = q'_4(\pi(R_1, R_2, T_3)) = q'_4(R_2, T_3, R_1)$ for permutation $\pi = (3, 1, 2)$. Thus, $(\mathcal{S}_3[1], \mathcal{S}_3[2], \mathcal{S}_3[3]) \mapsto (\mathcal{S}_4[3], \mathcal{S}_4[1], \mathcal{S}_4[2])$ is a pattern-preserving mapping.

2.2 Why we need Dissociation

We next show queries that are *logically equivalent* and have the *same table signature*, but they have *different relational patterns*. This example motivates why we defined relational patterns based on the dissociated signature of a query.

Example 8 (Different patterns). *Consider table $R(A, B)$ and the two Datalog queries:*

$$\begin{aligned} Q^1_5(x) &:\!- R(x, \_), R(x, \_). \\ Q_6(x) &:\!- R(x, y), R(\_, y). \end{aligned}$$

Both queries are logically equivalent to $Q(x) :\!- R(x, \_)$ and thus also logically equivalent to each other. However, they have different patterns: Q^1_5 never uses the second attribute of R, whereas Q_6 uses that attribute to join both occurrences of R. This difference becomes even more apparent in languages like SQL and TRC that use the named perspective: Q^1_5 is well-defined even if R is unary.

We next show how dissociation allows us to distinguish

[3] Notice that our definition of "dissociation" is influenced by its original use in probabilistic inference [15], yet it differs slightly.

[4] Quantification either happens explicitly (SQL) or implicitly (RA).

the two patterns. First, notice that both queries have two table references to R, and hence we need to associate each appearance to the "semantic role" it plays in the query. We achieve this by first dissociating the two occurrences of R into two fresh base tables (with the same schema):

$$Q_5^{1'}(x) :- R_1(x,\_), R_2(x,\_).$$
$$Q_6'(x) :- R_3(x,y), R_4(\_,y).$$

The dissociated queries encode functions $q_5^{1'}(R_1,R_2)$ and $q_6'(R_3,R_4)$, and it is easy to verify that neither of the two possible mappings $h_1=\{(R_1,R_3),(R_2,R_4)\}$ nor $h_2=\{(R_1,R_4),(R_2,R_3)\}$ preserves logical equivalence between them. For example, $Q_5^{1'}$ returns an answer over database $\{R_1(1,2),R_2(1,3)\}$, but Q_6' does not under either h_1 or h_2.

However, Q_5^1 is pattern-isomorphic to the TRC query

$$\{Q_5^2(A) \mid \exists r_1 \in R, r_2 \in R[q.A = r_1.A \wedge r_1.A = r_2.A]\}$$

To see that, notice that its dissociated query

$$\{Q_5^{2'}(A) \mid \exists r_1 \in R_5, r_2 \in R_6[q.A = r_1.A \wedge r_1.A = r_2.A]\}$$

encodes the function $q_5^{2'}(R_5,R_6)$ and allows the mapping $h_3=\{(R_1,R_5),(R_2,R_6)\}$ from $Q_5^{1'}$ to $Q_5^{2'}$, which preserves logical equivalence: $q_5^{1'}(R_1,R_2)=q_5^{2'}(R_1,R_2)$.

2.3 Queries that re-use Views

By design, our definition of patterns ignores views. To see why, consider a Datalog query that returns nodes in a graph that are the starting point of a 3-hop path:

$$Q_7^1(x) :- E(x,y), E(y,z), E(z,\_).$$

The following query uses the same logical pattern (find three edges that join and keep the start node), even though it defines an IDB I along the way:

$$I(y) :- E(y,z), E(z,\_).$$
$$Q_7^2(x) :- E(x,y), I(y).$$

Under Definition 6, Q_7^1 and Q_7^2 use the same pattern.

Also recall that Definition 5 requires IDBs to be used only once. Otherwise, we could define IDBs for each base table and re-use them in the query. Since we ignore IDBs, such a re-use would prevent us from understanding the semantic role of tables in the query from the signature of the base tables. Thus, to determine the pattern of a query with view re-use, we first need to clone the view definitions.

Example 9 (Patterns). *Consider the Datalog query*

$$H(x,y) :- R(x,y).$$
$$I(x,y) :- H(x,\_), T(y).$$
$$Q_8^1(x,y) :- H(x,z), \neg I(x,y).$$

The query uses the IDB H twice, and thus the computational flow represents a DAG and not a tree. To make it a tree, we can clone the view definition:

$$H_1(x,y) :- R(x,y).$$
$$H_2(x,y) :- R(x,y).$$
$$I(x,y) :- H_1(x,\_), T(y). \quad (7)$$
$$Q_8^2(x,y) :- H_2(x,z), \neg I(x,y).$$

Q_8^2 (7) has the same pattern as Q_4 (6).

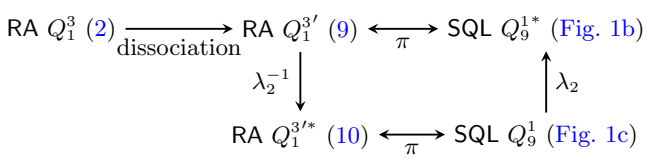


Figure 2: Weakly commutative diagram for Example 11 showing that the RA expression Q_1^3 (2) uses the same pattern as SQL expression Q_9^1 (Fig. 1c) because their dissociations are logically equivalent (shown here with π) after applying schema mapping λ_2. Since Q_9 is free, it is already dissociated.

2.4 Pattern Equivalence across Schemas

We next generalize the notion of pattern equivalence (or pattern isomorphism) of logically-equivalent queries to queries across *different schemas*.[5] The idea is that while two queries over different schemas can't be logically equivalent, they can become so after appropriately replacing schema elements (tables, attributes, constants). To make this mapping unambiguous, we define it over the dissociated queries.

Concretely, given dissociated queries Q_a' and Q_b', let λ be a *schema mapping* $\lambda : Q_a' \to Q_b'$ that replaces table names, attribute names, and constants from Q_a' with those from Q_b'. Denote $Q_a'^* = \lambda(Q_a')$ and its function $q_a'^*$.[6]

Definition 10 (Pattern isomorphism). *Two queries Q_a and Q_b have the same relational pattern if their dissociated queries Q_a' and Q_b' (with dissociated signatures $\mathcal{S}_a'$ and $\mathcal{S}_b'$) admit an invertible schema mapping $\lambda : Q_a' \to Q_b'$ under which $Q_a'^* = \lambda(Q_a')$ and Q_b' are logically equivalent, i.e., $q_a'^*(\mathcal{S}_a') = q_b'(\pi(\mathcal{S}_a'))$ for some permutation π.*

Example 11 (Example 1 continued). *Consider the queries Q_1^1 (1) and Q_2^1 (4) which are already dissociated (every table appears once). We get Q_1^{1*} via the schema mapping $\lambda_1 = \{$(Q$_1^1$.sname, Q$_1^{1*}$.sname), (Sailor.sid, SX.sno), (Sailor.sname, SX.sname), (Boat.bid, PX.pno), (Reserves.sid, SPX.sno), (Reserves.bid, SPX.pno)$\}$:*

$$\{Q_1^{1*}(sname) \mid \exists s \in SX[Q_1^{1*}.sname = s.sname \wedge \forall b \in PX[\exists r \in SPX[s.sno = r.sno \wedge r.pno = b.pno]]]\} \quad (8)$$

Notice that Q_1^{1*} (8) and Q_2^1 (4) are logically equivalent, and thus Q_1^1 (1) and Q_2^1 (4) use the same pattern.

Next, consider the SQL query Q_9^1 (Fig. 1c). We call it "free" to emphasize that all its tables and their attributes have distinct names—thus it is already dissociated. We show (Fig. 2) that Q_9^1 in SQL is pattern-isomorphic to Q_1^3 (2) in RA by applying to Q_9^1 the schema mapping $\lambda_2 =$ {(Q$_9^1$.att1, Q$_9^{1*}$.sname), (Table1.att2, Sailor1.sid), (Table1.att3, Sailor1.sname), (Table2.att4, Boat.bid), (Table3.att5, Reserves.sid), (Table3.att6, Reserves.bid), (Table4.att7, Sailor2.sid),} to get the SQL query Q_9^{1*} shown in (Fig. 1b). Notice that Q_9^{1*} and Q_1 are placeholder names for the output tables which are not shown in SQL—but

[5] Our SIGMOD'24 paper originally called this notion "pattern similarity," but we now believe keeping "equivalence" is more appropriate.

[6] Notice that this schema mapping is reversible (bijective) since the queries are dissociated. Also mappings between tables with different arities do not pose a problem: In the "named perspective" [28] (as in SQL and TRC) we only need to map between attributes that are used in the queries. In the "unnamed perspective" (as in DRC and Datalog), the mappings use positions with unused positions padded with anonymous variables, as in Q_1^4 (3).

their attributes are. This SQL expression Q_9^{1*} is logically equivalent to the RA query Q_1^3 from (2) after dissociation:

$$Q_1^{3'} = \pi_{sname}(Sailor_1 \triangleright \pi_{sid}((\pi_{sid} Sailor_2 \times \pi_{bid} Boat) - \pi_{sid,bid} Reserves)) \quad (9)$$

Similarly, in the other direction, the SQL expression Q_9^1 (Fig. 1c) is logically equivalent to the RA expression:

$$Q_1^{3'*} = \rho_{att1} \pi_{att3}(Table1 \triangleright_{att7=att3} \pi_{att7}((\pi_{att7} Table4 \times \pi_{att4} Table2) - \pi_{att5,att6} Table3)) \quad (10)$$

2.5 Limits of Visual Query Representations for understanding Relational Patterns

Our SIGMOD 2024 paper proves that there are queries in RC whose pattern cannot be represented in RA.

THEOREM 12 (Pattern separation). *There is a query in RC for which there is no pattern-isomorphic query in RA.*

When using visual query representations to help users understand a relational query, then the visual representation should be able to express all patterns in the source language. A survey [2] and extensive related work sections in our later work [8, 10, 13] show that most visual query representations build upon RA in that they model data flowing between relational operators. A consequence of our work is that such RA-based query visualizations have inherent limits in helping users understand the patterns of relational queries—particularly those with universal quantification, which is traditionally challenging to grasp [22, 27, 29].

COROLLARY 13. *Visual query representations that directly map RA operators to visual marks cannot represent all relational patterns from RC (and SQL).*

What one would need is a diagrammatic query representation that not only has the same logical expressiveness as RC but also the same pattern-expressiveness.

DESIDERATUM 14. *An effective visual query representation must have the logical expressiveness of RC **and** the same pattern expressiveness.*

3 Relational Diagrams

Relational Diagrams provide a diagrammatic representation of TRC, illustrating the semantic roles that table references have within a query. They can represent all relational patterns in the non-disjunctive fragment of relational languages.

3.1 Relational Diagrams in a nutshell

Relational Diagrams combine ideas from QueryVis [3, 11, 16] and Peirce's Existential Graphs [20, 23, 24].

❶ **Named UML perspective**: Like QueryVis, they adopt the named perspective and follow a familiar UML convention, representing tables as rectangular boxes with table names on top and attribute names below in separate rows (see Fig. 3). ❷ **Quantification over tuples**: As with QueryVis, depicted tables are existentially quantified. In contrast, Existential Graphs overload the meaning of lines, using them to denote quantification over domain values, which has led to considerable confusion (see [8, 13] for details). ❸ **No table aliases**: Like QueryVis and Datalog, they

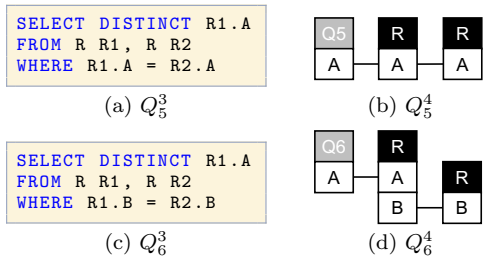


Figure 3: Queries Q_5 and Q_6 from Example 8 in SQL and as Relational Diagrams. While logically equivalent, the queries are not pattern-isomorphic, as is more apparent in the right column.

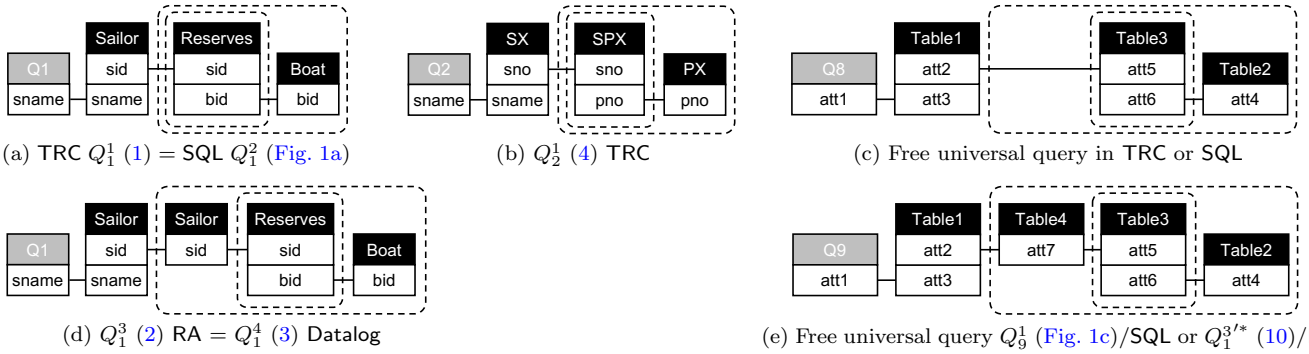
do not need table aliases, as attributes are displayed in proximity to the table they belong to. ❹ **Necessary attributes only**: Like QueryVis and TRC, they reduce clutter by showing only attributes that are used in the query. ❺ **Output table**: An explicit output table emphasizes the compositional nature of relational queries, where both the input and output are tables. ❻ **Familiar schema notation**: We believe that a simple conjunctive query should be shown with join conditions as lines connecting table attributes. Similar representations for database schemas are widely known from standard introductory database textbooks like [6, 26]. ❼ **Negation** is a key extension over relational schemas. Similar to Existential Graphs, we represent negation via explicitly nested negation scopes. This contrasts with QueryVis which overloads lines for representing nesting scopes. A negation scope is a closed path that partitions the canvas into a subcanvas that is negated (the enclosed part) and everything else that is not (the unenclosed part). We use dashed rounded rectangles (see Fig. 4). Recursive partitioning of the canvas allows us to represent a tree-based nesting order that corresponds to the nesting order of subqueries in SQL. We call the *main canvas* the root of that nesting hierarchy and each node a *partition of the canvas*.

3.2 SQL*: The non-disjunctive Fragment of SQL under Set Semantics

Finding diagrammatic representations for arbitrary disjunctions is surprisingly hard and has been an open question for some time [8, 24, 25]. We circumvent that problem by representing queries as unions of disjunction-free queries. The non-disjunctive fragment of TRC has a particularly natural and direct translation to and from Relational Diagrams. We describe that first and then add union later in Section 3.5.

SQL*. While the translation of relational queries into Relational Diagrams is formally defined from TRC, we present it here using SQL, a more widely recognized language. We first define SQL*, the non-disjunctive fragment of SQL under set semantics by the Extended Backus-Naur Form (EBNF) [19] grammar in Fig. 5. This fragment is interpreted under set semantics and binary logic (it prohibits NULL values in the base tables). To emphasize set semantics, we explicitly include the `DISTINCT` operator in all SQL statements.

Guarded predicates. We introduce an additional requirement that each predicate must be guarded, meaning it includes a *local attribute* whose table variable is bound within the scope of the last negation. A guarded predicate guarantees that it can be applied in the same negation scope as its local table variable. This requirement also prevents hidden disjunction. To illustrate, consider the SQL query in Fig. 6a.



(a) TRC Q_1^1 (1) = SQL Q_1^2 (Fig. 1a) (b) Q_2^1 (4) TRC (c) Free universal query in TRC or SQL

(d) Q_1^3 (2) RA = Q_1^4 (3) Datalog (e) Free universal query Q_9^1 (Fig. 1c)/SQL or $Q_1^{3\prime *}$ (10)/RA

Figure 4: Relational Diagrams for the universal queries Q_1 "*Find names of sailors who reserved all boats*" and Q_2 "*Find names of suppliers who supply all parts*" from Example 1, and Q_9 from Example 11. Relational Diagrams build upon UML notation for relational schemas and add a dashed rectangular box to show negation of the enclosed subquery. We see that the 3 queries in the first row use the same relational pattern; likewise the 2 queries in the second row. We show in our SIGMOD 2024 paper that RA (even if extended with an antijoin operator) needs 4 visual marks to express such a universal query. It follows that any visual query representation that maps the operators of RA to visual marks cannot explain the full range of patterns available in TRC or SQL, including the first row's pattern.

Q::=	SELECT [DISTINCT] (C {, C} | *) FROM R {, R} [WHERE P]	Non-Boolean query
	SELECT NOT (P)	Boolean query
|	SELECT [NOT] EXISTS (Q)	Boolean query
C::=	[T.]A	column or attribute
R::=	T [[AS] T]	table (table alias)
P::=	P {AND P}	conjunction of predicates
|	C O C	join predicate
|	C O V	selection predicate
|	NOT '('P')'	negation
|	[NOT] EXISTS '('Q')'	existential subquery
|	C [NOT] IN '('Q')'	membership subquery
|	C O (ALL '('Q')' | ANY '('Q')')	quantified subquery
O::=	= | <> | < | ≤ | ≥ | >	comparison operator
T::=		table identifier
A::=		attribute identifier
V::=		string or number

Figure 5: EBNF Grammar of SQL*: Statements enclosed in [ ] are optional; statements separated by | indicate a choice between alternatives; parentheses without quotation marks ( ) group alternative choices; parentheses with quotation marks '(' ')' form part of the test. Additionally, the main query requires the `DISTINCT` keyword (if non-Boolean), and all join and selection predicates need to be *guarded* (Definition 15).

```
SELECT DISTINCT R.A
FROM R
WHERE NOT EXISTS (
  SELECT *
  FROM S
  WHERE S.B = R.B
  AND R.A = 0)
```

(a)

```
SELECT DISTINCT R.A
FROM R
WHERE NOT EXISTS (
  SELECT *
  FROM S
  WHERE S.B = R.B)
OR R.A <> 0
```

(b)

Figure 6: Disjunctions can be hidden if predicates are not guarded. (See Section 3.2 and Definition 15 for explanation).

At first glance, it does not appear to contain a disjunction. However, the predicate "`R.A = 0`" can be pulled from the inner query, and by applying De Morgan's law, a disjunction emerges, as shown in Fig. 6b. To prevent both explicit and hidden disjunctions, the non-disjunctive fragment permits only conjunctions of guarded predicates.

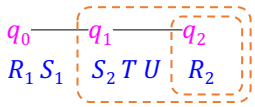
Definition 15 (Guarded predicate). *A predicate is guarded if it includes an attribute of a table variable that is bound within the same negation scope as the predicate.*

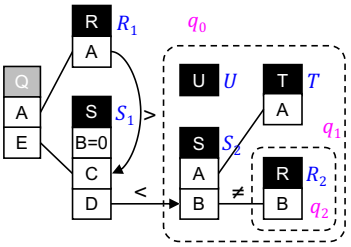
3.3 From SQL* to Relational Diagrams

We next illustrate the 5-step translation from any valid SQL* expression to its Relational Diagram by translating the SQL* query in Fig. 7a to the diagram in Fig. 7c. Notice that the translation hinges on three crucial conditions that the input fulfills: (1) quantifications in SQL are existential (universal quantifiers are replaced using De Morgan's law), (2) SQL* only allows conjunction between predicates, and (3) all predicates in SQL* are guarded (recall Definition 15).

❶ **Create canvas partitions**: The scopes of the negations in a SQL query are nested. We translate this *negation hierarchy* into a nested partition of the canvas (ours is in Fig. 7b). ❷ **Place tables**: Each table reference is placed in the canvas partition (negation scope) where it is bound. For example, the table references for table variables $R1$ and $S1$ are outside any negation scope and thus placed in the root partition q_0. Recall that, like Datalog and RA, Relational Diagrams do not need table aliases. ❸ **Place selection predicates**: Predicates within a partition are combined via conjunction. Thus, they can be added one after the other. Since all selection predicates are guarded, they can be correctly interpreted by placing them in the same partition as their respective table. Selection predicates are displayed inline with attributes, e.g. "`S1.B=0`" as "`B=0`" within the correct S table. ❹ **Place join predicates**: For each join predicate, we add the two attributes (if not already present) and connect them with an edge with any comparison operator added as a label. An attribute participating in multiple join predicates needs to be shown only once. Equijoins are the default, with no operator label shown. Asymmetric joins include an arrowhead at one end of the edge. Since guarded join predicates have either one or both attributes in a partition of a local table, the negation can be correctly interpreted. An example unguarded predicate would be "`not(R2.B = S2.B)`", which is equivalent to the predicate "`R2.B <> S2.B`". The latter is guarded as long as one of the two attributes is in the local scope of the last negation, as $R2$ is in our case. ❺ **Place and connect the output table**: The safety conditions for SQL imply that output predicates can only be chosen from tables outside all negations, thus in the root scope (our partition q_0). If the query is non-Boolean, we add a new table Q for the query with a unique gray background to show the difference from table references.

```
SELECT DISTINCT R1.A, S1.C AS E
FROM R AS R1, S AS S1
WHERE S1.B = 0
AND R1.A > S1.C
AND not exists
  (SELECT *
  FROM S AS S2, T, U
  WHERE S2.B > S1.A
  AND S2.A = T.A)
  AND not exists
    (SELECT *
    FROM R AS R2
    WHERE R2.B <> S2.B))
```

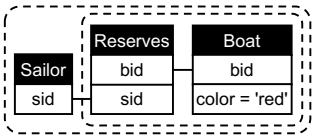




(a) Example query in SQL* (b) Negation hierarchy (c) Relational Diagram

Figure 7: Section 3.3: Example SQL* query (a), negation hierarchy with table references (b), and corresponding Relational Diagram (c). Colored partitions q_i (purple) and table variables R_j (blue) are not part of the diagram and shown only to discuss the correspondence.

```
SELECT NOT EXISTS
 (SELECT *
 FROM Sailor s
 WHERE NOT EXISTS
  (SELECT b.bid
  FROM Boat b, Reserves r
  WHERE b.color='red'
  AND r.bid=b.bid
  AND r.sid=s.sid))
```



(a) (b)

Figure 8: Example 17: "*All sailors reserve a red boat.*"

We denote the non-disjunctive fragment of Relational Diagrams discussed so far as Relational Diagrams*. Our paper also defines conditions for a Relational Diagram* to be valid, gives a translation from a Relational Diagram* to TRC* or SQL*, and shows that every valid SQL* query has a pattern-isomorphic representation as a Relational Diagram*.

THEOREM 16 (Pattern-expressiveness of Relational Diagrams*). *Every SQL* query can be represented as a valid and pattern-isomorphic Relational Diagram*. Every valid Relational Diagram* has an unambiguous and pattern-isomorphic interpretation in SQL*.*

3.4 Logical constraints and sentences

Boolean queries are formulas without free variables and without an output table. They represent relational sentences (constraints) that are true or false. Being able to express them allows us to compare our formalism against a long history of formalisms for logical statements [8]. We give an intuitive example, with more examples shown in [13].

EXAMPLE 17. *Consider the statement* "All sailors reserve a red boat" *in Fig. 8. The first 4 steps of the translation in Section 3.3 from the SQL statement in Fig. 8a still work even though it has no FROM clause before the first NOT. As a result, the root canvas in Fig. 8b (q_0, though unlabeled here) remains empty. Importantly, Definition 6 on query pattern isomorphism still works as it is defined based on the base tables.*

3.5 Relational completeness

To make Relational Diagrams relationally complete, we next remove the earlier restrictions and allow disjunctions and unions. We also add a visual device to Relational Diagrams that achieves logical equivalence to the other relational query languages. Unfortunately, equivalence relies on transformations, which implies that Relational Diagrams can no longer represent all patterns of queries with disjunctions.

The syntactic device we add is inspired by the representation of disjunction in Datalog. It was also proposed by Peirce in his discussion of Euler diagrams [20, page 4.366] (see also [25, sect. 2.3.1]): we allow placing several Relational Diagrams* on the same canvas, each in a separate *union cell*. Each cell then represents only conjunctive information (with negation), yet the relation among the different cells is disjunctive (a union of the outputs).

We next give two examples of logical transformations that are not pattern-preserving but that ensure relational completeness. Together with union cells, these transformations make Relational Diagrams *relationally complete*: every query expressible in full RA, safe TRC, Datalog¬, or our prior SQL* fragment extended by union and disjunctions of predicates[7] can then be represented as a logically-equivalent Relational Diagram. The first example shows how to avoid disjunctions if they are not at the root level. The second shows how to replace disjunctions in the root by unions of queries.

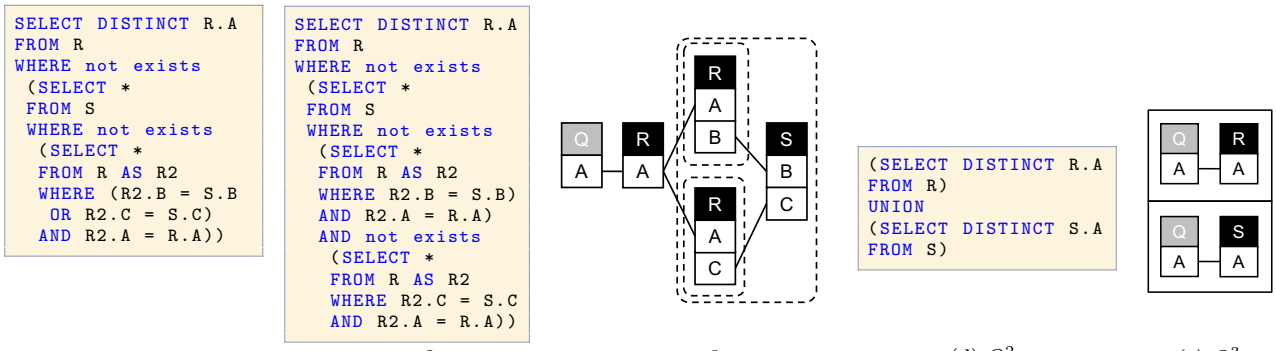
EXAMPLE 18 (Replacing disjunctions). *Consider the SQL query Q^1_{10} from Fig. 9a which contains a disjunction and is not in SQL*. Using De Morgan's law with quantifiers $\neg\exists r \in R[A \vee B] = \neg(\exists r \in R[A] \vee \exists r \in R[B]) = \neg\exists r \in R[A] \wedge \neg\exists r \in R[B]$, we can first reformulate the conditions including disjunction as DNF, and then distribute the quantifier over the conjuncts. This leads to a disjunction-free query Q^2_{10} but increases the number of table references. We also show this query as a logically-equivalent SQL* query (Fig. 9b) and Relational Diagram (Fig. 9c).*

EXAMPLE 19 (Union of queries). *Consider two unary tables $R(A)$ and $S(A)$ and the TRC query*

$$\{Q^1_{11}(A) \mid \exists r \in R[q.A = r.A] \vee \exists s \in S[q.A = s.A]\}$$

Figure 9d shows it as pattern-isomorphic SQL query, and Fig. 9e shows it as a Relational Diagram with two separate Relational Diagrams, each in a separate union cell, and each with the same attribute signature in the output table. This query cannot be written without the union operator in RA or SQL, nor Relational Diagrams* without union cells.*

[7] Extend the grammar from Fig. 5 with one additional rule: P::= '('P OR P')' and make adjustments to allow the UNION clause at the root between non-Boolean queries.



(a) Q^1_{10} (b) Q^2_{10} (c) Q^3_{10} (d) Q^2_{11} (e) Q^3_{11}

Figure 9: Illustrations for Example 18 on replacing disjunctions: (a) SQL with disjunctions, (b) logically-equivalent (yet not pattern-equivalent) SQL* statement, and (c) Relational Diagrams. Illustrations for Example 19 on creating the union of queries: (d) SQL with disjunctions, (e) logically-equivalent union of SQL* statements, and (f) Relational Diagram with union cells.

THEOREM 20 (Completeness). *Relational Diagrams with union cells are relationally complete. Thus, they can represent any query or statement in the safe fragment of TRC.*

The proof is in the full paper [13] and uses the earlier proven logical expressiveness of Relational Diagrams* and the fact that disjunctions can either be rewritten with De Morgan's law or pushed to the root. It also immediately follows that Relational Diagrams* (without union cells) can already express any logical statement expressible in TRC.

COROLLARY 21 (Completeness). *Any logical statement expressible in safe TRC can be represented by a logically-equivalent Relational Diagram* (without union cells).*

3.6 Two Applicability Studies

❶ **Can Relational Diagrams represent common patterns?** To answer this question, we mined 5 popular textbooks for example queries in RC, finding 58. We then determined the fraction that could be represented with the same pattern in Relational Diagrams (95%), QueryVis [3] (90%), QBE [32] (83%), RA (81%), and Datalog (80%).

❷ **Are Relational Diagrams useful for seeing patterns?** Through an IRB-approved, preregistered controlled experiment on Amazon Mechanical Turk (MTurk) [1], we found that users could distinguish 4 query patterns *faster* and *more accurately* using Relational Diagrams than SQL. The patterns are variations on the 4 Categorical Propositions from logic (some/none/all/not all) [30], written as queries over different schemas, such as: "*Find sailors who have reserved* (some/not any/not all/all) *boat(s).*"

After a brief tutorial, 50 participants each answered 32 questions asking them to choose which of the 4 patterns was shown. The questions used 32 different schemas, were parameterized, and alternated between Relational Diagrams and SQL. In short, our study showed:

RESULT 1 (Speed). *Participants were meaningfully faster at identifying patterns using Relational Diagrams than SQL: the median ratio of time Relational Diagrams/SQL = 0.70, 95% CI [0.63, 0.77].*

This aligns with prior studies finding that diagrammatic representations can help faster query understanding [16, 18].

RESULT 2 (Learning). *Participants got meaningfully faster during the study: median ratio of time 1st half/2nd half = 0.71, 95% CI [0.63, 0.79] for Relational Diagrams, = 0.70, 95% CI [0.51, 0.79] for SQL.*

Participants could learn how to read the patterns under repeated exposures with different schemas! This suggests that teaching based on common query patterns could be effective for more ready understanding and re-use later, just as we teach programming using software design patterns.

RESULT 3 (Accuracy). *Participants were considerably more often correct with Relational Diagrams than with SQL: mean difference in accuracy Relational Diagrams − SQL = 21%, 95% CI [13%, 29%].*

Not only are users faster with Relational Diagrams–and after only minimal training—they were more accurate as well!

All materials for our studies are available in our supplemental material [13]. The SIGMOD 2024 ARI checked that our collected data and analysis code was reproducible, and they also verified we provided sufficient code and documentation for all aspects of our study. The full paper [12] and its supplemental material [13] include informative visualizations of participant performance and further study details.

4 Future Work

Currently, Relational Diagrams cannot represent arbitrary disjunctive patterns. Can this limitation be overcome by a better diagram design? A long line of literature on limits of diagrams to express disjunctive information [24, 25] suggests that such an extension would require adding non-diagrammatic abstractions. Recent progress suggests that such an extension may ultimately be feasible [9]. An open question is whether diagrammatic abstractions can be *meaningfully* extended to the full set of relational transformations, including grouping and aggregation.

Acknowledgements. We thank the reviewers of our SIGMOD 2024 paper, the SIGMOD 2024 ARI reviewers, and our colleagues and friends at Northeastern's DATA Lab for their valuable feedback. This work was supported in part by a Khoury seed grant and the National Science Foundation (NSF) under award numbers IIS-1762268, IIS-2145382, and IIS-1956096. It was conducted in part while WG was visiting the Simons Institute for the Theory of Computing.

References

[1] Amazon Mechanical Turk (MTurk). https://www.mturk.com, 2023.

[2] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: a survey. *J. Vis. Lang. Comput.*, 8(2):215–260, 1997. DOI: 10.1006/jvlc.1997.0037.

[3] J. Danaparamita and W. Gatterbauer. QueryViz: helping users understand SQL queries and their patterns. In *EDBT*, pages 558–561. ACM, 2011. DOI: 10.1145/1951365.1951440.

[4] H. Darwen and C. J. Date. The third manifesto. *SIGMOD Rec.*, 24(1):39–49, Mar. 1995. DOI: 10.1145/202660.202667.

[5] C. J. Date. *An introduction to database systems*. Pearson/Addison Wesley Longman, 8th edition, 2003. URL: https://dl.acm.org/doi/10.5555/861613.

[6] R. Elmasri and S. Navathe. *Fundamentals of database systems*. Addison Wesley, 7th edition, 2015. URL: https://dl.acm.org/doi/book/10.5555/2842853.

[7] J. H. Gallier. *Discrete mathematics*. Universitext. Springer, 2011. URL: https://doi.org/10.1007/978-1-4419-8047-2.

[8] W. Gatterbauer. A comprehensive tutorial on over 100 years of diagrammatic representations of logical statements and relational queries. In *ICDE*, pages 5387–5392. IEEE, 2024. DOI: 10.1109/ICDE60146.2024.00407. Tutorial page: https://northeastern-datalab.github.io/diagrammatic-representation-tutorial/, Slides: https://northeastern-datalab.github.io/diagrammatic-representation-tutorial/ICDE_2024-Diagrammatic-Representations-Tutorial.pdf.

[9] W. Gatterbauer. A principled solution to the disjunction problem of diagrammatic query representations, 2024. DOI: 10.48550/ARXIV.2412.08583.

[10] W. Gatterbauer. A tutorial on visual representations of relational queries. *PVLDB*, 16(12):3890–3893, 2023. DOI: 10.14778/3611540.3611578. Tutorial page: https://northeastern-datalab.github.io/visual-query-representation-tutorial/.

[11] W. Gatterbauer. Databases will visualize queries too. *PVLDB*, 4(12):1498–1501, 2011. DOI: 10.14778/3402755.3402805. Recorded Talk: https://www.youtube.com/watch?v=kVFnQRGAQls.

[12] W. Gatterbauer and C. Dunne. On the reasonable effectiveness of relational diagrams: explaining relational query patterns and the pattern expressiveness of relational languages. *Proc. ACM Manag. Data*, 2(1):61:1–61:27, 2024. DOI: 10.1145/3639316.

[13] W. Gatterbauer and C. Dunne. Supplemental material for "On the reasonable effectiveness of relational diagrams", 2023. Homepage: https://relationaldiagrams.com/. Main supplemental material folder on OSF: https://osf.io/q9g6u/. Online appendix with all proofs, further illustrations, and study materials: https://arxiv.org/pdf/2401.04758. Textbook analysis: https://osf.io/u7c4z. User study tutorial: https://osf.io/mruzw. Stimuli-generating code: https://osf.io/kgx4y. The stimuli: https://osf.io/d5qaj. Stimuli/schema index CSV: https://osf.io/u8bf9. Stimuli/schema index JSON: https://osf.io/sn83j. Server code for hosting the study: https://osf.io/suj4a. Collected data: https://osf.io/8vm42. Executed user study analysis code: https://osf.io/f2xe3. Preregistered user study: https://osf.io/4zpsk/. Results from SIGMOD 2024 ARI (Availability & Reproducibility Initiative): https://reproducibility.sigmod.org/reports.html.

[14] W. Gatterbauer, C. Dunne, H. V. Jagadish, and M. Riedewald. Principles of query visualization. *IEEE Data Eng. Bull.*, 45(3):47–67, 2022. URL: http://sites.computer.org/debull/A22sept/p47.pdf.

[15] W. Gatterbauer and D. Suciu. Oblivious bounds on the probability of Boolean functions. *TODS*, 39(1):5:1–5:34, 2014. DOI: 10.1145/2532641.

[16] A. Leventidis, J. Zhang, C. Dunne, W. Gatterbauer, H. V. Jagadish, and M. Riedewald. QueryVis: logic-based diagrams help users understand complicated SQL queries faster. In *SIGMOD*, pages 2303–2318. ACM, 2020. DOI: 10.1145/3318464.3389767.

[17] Merriam-Webster.com. Pattern, 2025. URL: https://www.merriam-webster.com/dictionary/pattern.

[18] D. Miedema and G. Fletcher. SQLVis: visual query representations for supporting SQL learners. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9. IEEE, 2021. DOI: 10.1109/VL/HCC51201.2021.9576431.

[19] R. E. Pattis. EBNF: a notation to describe syntax. https://ics.uci.edu/~pattis/misc/ebnf2.pdf, 2013. (accessed on September 21, 2021).

[20] C. S. Peirce. *Collected Papers*. C. Hartshorne and P. Weiss, editors, volume 4. Harvard University Press, 1933. DOI: 10.1177/000271623417400185.

[21] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, USA, 3rd edition, 2002. URL: https://dl.acm.org/doi/book/10.5555/560733.

[22] P. Reisner. Human factors studies of database query languages: a survey and assessment. *ACM Comput. Surv.*, 13(1):13–31, 1981. DOI: 10.1145/356835.356837.

[23] D. D. Roberts. The existential graphs. *Computers & Mathematics with Applications*, 23(6):639–663, 1992. DOI: 10.1016/0898-1221(92)90127-4.

[24] S.-J. Shin. *The Iconic Logic of Peirce's Graphs*. The MIT Press, 2002. DOI: 10.7551/mitpress/3633.001.0001.

[25] S.-J. Shin. *The Logical Status of Diagrams*. Cambridge University Press, 1995. DOI: 10.1017/CBO9780511574696.

[26] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 7th edition, 2020. URL: https://www.db-book.com/db7/index.html.

[27] J. C. Thomas. Quantifiers and question-asking. Technical report, IBM Res. Rep. RC 5866, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, 1976. URL: https://apps.dtic.mil/sti/tr/pdf/ADA043032.pdf.

[28] J. D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I*. Computer Science Press, Inc., 1988. URL: https://dl.acm.org/doi/book/10.5555/42790.

[29] C. Welty and D. W. Stemple. Human factors comparison of a procedural and a nonprocedural query language. *ACM Trans. Database Syst.*, 6(4):626–649, 1981. DOI: 10.1145/319628.319656.

[30] Wikipedia contributors. Categorical proposition — Wikipedia, the free encyclopedia, 2025. URL: https://en.wikipedia.org/wiki/Categorical_proposition. [Online; accessed March-2025].

[31] Wikipedia contributors. Software design pattern — Wikipedia, the free encyclopedia, 2025. URL: https://en.wikipedia.org/wiki/Software_design_pattern. [Online; accessed February-2025].

[32] M. M. Zloof. Query-by-example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977. DOI: 10.1147/sj.164.0324.