On The Reasonable Effectiveness of Relational Diagrams[*]

Explaining Relational Query Patterns and the Pattern Expressiveness of Relational Languages

WOLFGANG GATTERBAUER, Northeastern University, USA
CODY DUNNE, Northeastern University, USA

Comparing relational languages by their logical expressiveness is well understood. Less well 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? To the best of our knowledge, we provide the first semantic definition of relational query patterns by using a variant of structure-preserving mappings between the relational tables of queries. This formalism allows us to analyze the *relative pattern expressiveness* of relational language fragments and create a hierarchy of languages with equal logical expressiveness yet different pattern expressiveness. Notably, for the non-disjunctive language fragment, we show that relational calculus can express a larger class of patterns than the basic operators of relational algebra.

Our language-independent definition of query patterns opens novel paths for assisting database users. For example, these patterns could be leveraged to create visual query representations that faithfully represent query patterns, speed up interpretation, and provide visual feedback during query editing. As a concrete example, we propose Relational Diagrams, a complete and sound diagrammatic representation of safe relational calculus that is provably (*i*) unambiguous, (*ii*) relationally complete, and (*iii*) able to represent all query patterns for unions of non-disjunctive queries. Among all diagrammatic representations for relational queries that we are aware of, ours is the only one with these three properties. Furthermore, our anonymously preregistered user study shows that Relational Diagrams allow users to recognize patterns meaningfully faster and more accurately than SQL.

CCS Concepts: • **Information systems** → **Relational database query languages**.

Additional Key Words and Phrases: Visual Query Representation, Logical Diagrams, Peirce Existential Graphs

ACM Reference Format:
Wolfgang Gatterbauer and Cody Dunne. 2024. On The Reasonable Effectiveness of Relational Diagrams: Explaining Relational Query Patterns and the Pattern Expressiveness of Relational Languages. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 61 (February 2024), 27 pages. https://doi.org/10.1145/3639316

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 for decades, and formalisms for comparing expressiveness are well-developed and understood. We do not yet have a similarly developed machinery to reason

[*]The title is a reference to Wigner's 1960 article [55] in which he states that "the enormous usefulness of mathematics in the natural sciences is something bordering on the mysterious and that there is no rational explanation for it." While there have been similar observations of surprising effectiveness for both data [36] and logic [37] in our community, our strong experimental evidence of Relational Diagrams helping users understand relational patterns better is actually quite expected.

Authors' addresses: Wolfgang Gatterbauer Northeastern University, USA, w.gatterbauer@northeastern.edu; Cody Dunne Northeastern University, USA, c.dunne@northeastern.edu.

2836-6573/2024/2-ART61
https://doi.org/10.1145/3639316

about *relational query patterns*. Intuitively, a query pattern should capture "a certain way of writing a query." To be universally applicable, a formalization would have to be applicable across the four major relational languages—Datalog, Relational Algebra (RA), Relational Calculus (RC), and SQL—and thus be orthogonal to questions of syntax and procedural or declarative language design.

We posit that identifying patterns in queries could open novel paths for assisting users [33], especially learners who try to understand the structure behind relational queries written in different languages. It could help learners spot similarities in queries across different schemas and thus more easily separate intent (the logic) from the particular syntactic expression. On an even more fundamental level, establishing a separation on "pattern expressiveness" between relational languages could lead to new insights into the intrinsic properties of relational languages and algebraic limits of visualizations. An important insight that we establish in this paper is that visual languages which build upon *the operators of RA cannot faithfully express all query patterns*, and instead necessitate reformulating queries and thus changing their patterns.

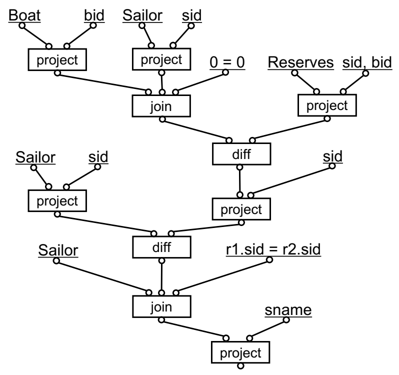


Fig. 1. DFQL [10] visualization of the TRC query from Example 1. Notice the 3 instances of the Sailor relation and thus a different "structure" of the visualization from the original query.

Example 1 (Understanding the structure of a TRC query). *Imagine Kiyana, a theory-leaning undergraduate student, trying to understand relational query languages better. Kiyana has been reading the chapters on relational calculus across several books. In the textbook by Ramakrishnan and Gehrke [44] (page 121 of Sect. 4.3.1) she finds the query* "(Q9) Find the names of sailors who have reserved all boats" *written in "tuple relational calculus" as follows:*

$$\{P \mid \exists S \in Sailor\, \forall B \in Boat (\exists R \in Reserves \\ (S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\} \tag{1}$$

She tries to understand "the structure" of the query and translates it first into RA, and then from there into DFQL (Dataflow Query Language) [10, 14, 35].[1] *DFQL is a visual representation that is relationally complete by mapping its visual symbols to the operators of RA. Kiyana quickly notices that she cannot translate the query into RA without using* additional *Sailor relations.*

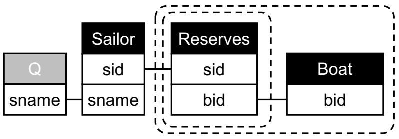
$$Q = \pi_{sname}\big(Sailor \bowtie \big(\pi_{sid}Sailor - \pi_{sid}\big((\pi_{sid}Sailor \times \pi_{bid}Boat) - \pi_{sid,bid}Reserves\big)\big)\big)$$

As a result, she does not find the resulting DFQL visualization (Fig. 1) very helpful because there is an obvious mismatch in "its structure" with 3 instances of Sailor relations. She wonders whether she is missing an obvious simpler translation into RA or whether there is none. As is, she does not find this query visualization very helpful.

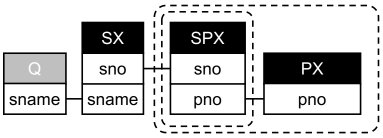
As a consequence, *no query visualization that relies on the operators of RA* could help Kiyana with what she would like to see: a simple visual representation that captures the structure of the query as written in its original logical form.

Example 2 (Comparing RC queries from textbooks). *Kiyana continues looking through different textbooks and finds in Date's textbook [19] (page 224 of Sect. 8.3) the query* "8.3.6 Get supplier names

[1]DFQL is one of several visual query languages mentioned as relationally complete in a widely cited survey [14]. Kiyana found a detailed online documentation [35] and worked out examples [30].



(a) q_1: "Find names of sailors who reserved all boats."



(b) q_2: "Find names of suppliers who supply all parts."

Fig. 2. Relational Diagrams representating the two queries from Example 1 ([44]) and Example 2 ([19]). Notice the clearly discernible similar "relational query patterns."

for suppliers who supply all parts" *written in "tuple calculus" as follows:*

```
SX.NAME WHERE NOT EXISTS PX (NOT EXISTS SPX
           SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO))
```
(2)

From the natural language description, the query seems to follow a similar pattern as the earlier one ("Return X which have a relationship with all Y"). But that apparent similarity is difficult to see from the two expressions. She wonders whether there is a simple way to see that those two queries somehow follow a "similar structure."

In this paper, we show that there is indeed a simple and arguably-natural diagrammatic representation (also called visualization, in short) that allows Kiyana to (*i*) represent her queries in a way that preserves their logical structure (or pattern), (*ii*) decide whether two logically-equivalent queries have the same pattern, and (*iii*) see whether any two queries, even across different schemas, use a "similar pattern." We call this visualization Relational Diagrams. [47]. See Fig. 2 and notice how every relation from the two queries maps to exactly one relation in the Relational Diagrams. Also, notice how the similar structure of both queries becomes natural to see.

Our 1st contribution: *query patterns.* We develop a precise language-independent notion of relational query pattern 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 which are inherently language-dependent). Intuitively, our formalism reasons about mappings between the (existentially or universally) quantified relations referenced in two queries. Yet it is not trivial to turn this intuition into a working definition that can be applied to any relational query and language (we include examples to show that seemingly easier mapping definitions would fail on queries). We believe that this idea is the "right" approach for defining relational pattern and show how to use it to compare relational query languages by their abilities 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 an arguably simple and intuitive diagrammatic representation of relational queries called Relational Diagrams [32] and prove that (*i*) it is unambiguous (every diagram has a unique logical interpretation), (*ii*) it is relationally complete (every relational query can be expressed in a logically-equivalent Relational Diagram), and (*iii*) that it can express all query patterns in the non-disjunctive fragment of relational query languages and those with union at the root. In particular, we prove that no prior or future diagrammatic representation based on RA could represent all relational query patterns from RC. Our user study (Section 6.2) shows that our formalisms helps users recognize patterns faster than with SQL.

Outline of the paper. Section 2 defines the *non-disjunctive fragment* of relational query languages for Datalog, Relational Algebra (RA), Tuple Relational Calculus (TRC), and SQL, and proves that they have equivalent logical expressiveness. Section 3 shows that the non-disjunctive fragment allows for an arguably natural diagrammatic representation system that we term Relational Diagrams*(with a star). We give the formal translation from the non-disjunctive fragment of TRC to Relational Diagrams* and back, and define their formal validity. We use Relational Diagrams* for the remainder of the paper to illustrate "query patterns." Section 4 formalizes the notion of a relational query pattern and contributes a novel hierarchy of pattern expressiveness among the above four languages for the non-disjunctive fragment. We prove that Relational Diagrams* have strong structure-preserving properties in that they can express all query patterns in this fragment. Section 4.4 formalizes "similar patterns" across different schemas. This extended notion allows us to see similarities across queries that use different relations and are thus not logically equivalent. Section 5 adds a single visual element (called a *union cell*) to Relational Diagrams* to make the resulting Relational Diagrams relationally complete.[2] We also show that even without that element, Relational Diagrams*can express all logical statements of first-order logic. This allows us to compare our diagrammatic formalism against a long history of diagrams for representing logical sentences. Section 6 includes two studies. The first shows that more logical queries across five popular textbooks have pattern-isomorphic representations in Relational Diagrams than either RA, Datalog, QBE, or QueryVis. The second controlled user experiment study that using Relational Diagrams instead of SQL helps users recognize patterns across different schemas faster and more often correctly. Section 7 contrasts our formalism with selected related work. In particular, we discuss the connection to Peirce's existential graphs [43, 48, 50] and show that our formalism is more general and solves interpretational problems of Peirce's graphs which have been the focus of intense research for over a century.

Due to space constraints, we had to move proofs, several intuitive illustrating examples, all study details, and more detailed comparison against related work to the online appendix [32].

2 THE NON-DISJUNCTIVE FRAGMENT OF RELATIONAL QUERY LANGUAGES

This section defines the *non-disjunctive fragment* of relational query languages. Throughout, we assume a linear order over the active domain and thus explicitly allow built-in predicates using ordered operators such as $<$, in addition to equality $=$ and disequality $\neq$.

We assume the reader to be familiar with Datalog$^{\neg}$ (non-recursive Datalog with negation), RA (Relational Algebra), TRC (safe Tuple Relational Calculus), SQL (Structured Query Language), and the necessary safety conditions for TRC and Datalog$^{\neg}$ to be equivalent in logical expressiveness to RA. We also assume familiarity with concepts such as relations, predicates, atoms, and the named and unnamed perspective of relational algebra. The most comprehensive exposition of these topics we know of is Ullman's 1988 textbook [54], together with resources for translating between SQL and relational calculus [12, 20]. These connections are also discussed in most database textbooks [25, 28, 44, 51], though in less detail. We only cover TRC and not Domain Relational Calculus (DRC) as the 1-to-1 correspondence between DRC and TRC is straight-forward [25, 51], and—as we will discuss in Section 7.1—TRC has a more natural translation into diagrams than DRC.

2.1 Non-recursive Datalog with negation

We start with Datalog since the definition is most straightforward. Datalog expresses disjunction (or union) by repeating an Intensional Database (IDB) predicate in the head of multiple rules. For

[2]Although disjunctions can be composed of conjunction and negation using De Morgan's law ($A \vee B = \neg(\neg A \wedge \neg B)$), this additional visual symbol is necessary: for safe relational queries, DeMorgan is not enough, as there is no way to write a safe Tuple Relational Calculus (TRC) expression "*Return all entries that appear in either R or S*" that avoids a union operator. This is part of the textbook argument for the union operator being an essential, non-redundant operator for relational algebra.

example, consider the following query in Datalog:

$$\begin{aligned} Q(x) &:- R(x,y), S(x), T(\_), y > 5. \\ Q(x) &:- R(x,y), S(\_), T(x), y > 5. \end{aligned} \tag{3}$$

The underscore stands for a variable that appears only once [28]. This query cannot be expressed without defining at least one IDB at least twice, in our case the result table $Q(x)$. This leads to a natural definition of the non-disjunctive fragment of Datalog$^\neg$:

Definition 1 (Datalog*). Non-disjunctive non-recursive Datalog with negation (Datalog*) is the non-recursive fragment of Datalog$^\neg$ with built-in predicates where every IDB appears in the head of exactly one rule and can be used maximally once in any body.

Notice that Datalog* inherits all restrictions from non-recursive Datalog$^\neg$ with built-in predicates [54], and thus rules out the existence of an IDB in both the head and the body of the same rule. The restriction of IDB's being used maximally once rules out views to be used multiple times (including simple copies of input tables).

2.2 Relational Algebra (RA)

We focus on the subfragment of basic RA $(\times, \sigma, \bowtie_c, \pi, -)$ that contains no union operator $\cup$ and in which all selection conditions are simple (i.e. they do not use the disjunction operator $\vee$). A simple condition is $C = (X \theta Y)$ where X is an attribute, Y is either an attribute or a constant, and θ is a comparison operator from $\{=, \neq, <, \leq, >, \geq, \}$. Notice that conjunctions of selections can be modeled as concatenation of selections, e.g., $\sigma_{C_1 \wedge C_2}(R)$ is the same as $\sigma_{C_1}(\sigma_{C_2}(R))$. The Datalog$^\neg$ query from (3) cannot be expressed in that fragment. Assuming the schema $R(A,B)$, $S(A)$, $T(A)$, a translation either requires the disjunction operator $\vee$ as in:

$$\pi_A\big(\sigma_{A=C \vee A=D}\big(\sigma_{B>5}(R) \times \rho_{A\to C}(S) \times \rho_{A \to D}(T)\big)\big)$$

or the union operator $\cup$ as in:

$$\pi_A\big(\sigma_{B>5}(R) \bowtie S \times \rho_{A\to D}(T)\big) \cup \pi_A\big(\sigma_{B>5}(R) \bowtie T \times \rho_{A \to C}(S)\big)$$

Definition 2 (RA*). The non-disjunctive fragment of basic Relational Algebra (RA*) results from disallowing the union operator $\cup$ and by restricting selections to conjunctions of simple predicates.

2.3 Tuple Relational Calculus (TRC)

Recall that some variants of safe TRC only allows existential quantification (and not universal quantification) [54]. Predicates are either join predicates "$r.A\,\theta\,s.B$" or selection predicates "$r.A\,\theta\,v$", with r, s being table variables and v a domain value. WLOG, every existential quantifier can be pulled out as early as to either be at the start of the query, or directly following a negation operator. For example, instead of $\neg(\exists r \in R[r.A = 0 \wedge \exists s \in S[s.B = r.B]])$ we rather write this sentence canonically as $\neg(\exists r \in R, s \in S[r.A = 0 \wedge s.B = r.B])$. This *canonical representation* implies that a set of existential quantifiers is always preceded by the negation operator, except for the table variables outside any scope of negation operators. Also, WLOG, we only allow equality conditions with the result table. For example, instead of $\{q(A) \mid \exists r \in R, s \in S[q.A = r.A \wedge s.A > q.A])]\}$ we rather write $\{q(A) \mid \exists r \in R, s \in S[q.A = r.A \wedge s.A > r.A])]\}$. Recall that at least one equality predicate for each output attribute is required due to standard safety conditions [54].

We will define an additional requirement that each predicate contains a local (or what we refer to as *guarded*) attribute whose table is quantified within the scope of the last negation. For example, we do not allow $\neg(\exists r \in R[\neg(r.A = 0)])$ because the table variable r is defined outside the scope of the most inner negation around the predicate $r.A = 0$. However, we allow the logically-equivalent

$\neg(\exists r \in R[r.A \neq 0])$ where the table variable r is existentially quantified within the same scope as the attribute $r.A \neq 0$.

Definition 3 (Guarded predicate). A predicate is *guarded* if it contains at least one attribute of a table that is existentially quantified inside the same negation scope as that predicate.

Intuitively, guarding a predicate guarantees that the predicates can be applied in the same logical scope where a table is defined. This requirement also avoids a hidden disjunction. To illustrate, consider the following TRC query:

$$\{q(A) \mid \exists r \in R[q.A = r.A \wedge \neg(\exists s \in S[r.A = 0 \wedge s.B = r.B])]\}$$

This query contains no apparent disjunction, however the predicate "$r.A = 0$" could be pulled outside the negation, and after applying De Morgan's law on the expression we get a disjunction:

$$\{q(A) \mid \exists r \in R[q.A = r.A \wedge \big(r.A \neq 0 \vee \neg(\exists s \in S[s.B = r.B])\big)]\}$$

To avoid both disjunctions and "hidden disjunctions", the non-disjunctive fragment *only allows conjunctions of guarded predicates*:

Definition 4 (TRC*). The non-disjunctive fragment of safe TRC (TRC*) disallows both universal quantification and disjunctions, and restricts predicates to conjunctions of *guarded predicates*.

In order to express the Datalog$^\neg$ query from (3) we need the disjunction operator. A possible translation is:

$$\{q(A) \mid \exists r \in R, s \in S, t \in T[q.A{=}r.A \wedge r.B{>}5 \wedge (r.A{=}s.A \vee r.A{=}t.A)]\}$$

2.4 SQL under set semantics

Structured Query Language (SQL) uses bag instead of set semantics and uses a ternary logic with NULL values. In order to treat SQL as a logical query language, we assume binary logic and no NULL values in the input database. It has been pointed out that "SQL's logic of nulls confuses people" and even programmers tend to think in terms of the familiar two-valued logic [53]. Our focus here is devising a general formalism to capture logical query patterns across relational languages, not on devising a visual representation of SQL's idiosyncrasies. To emphasize the set semantic interpretation, we write the DISTINCT operator in all our SQL statements.

We define the non-disjunctive fragment of SQL as the Extended Backus–Naur form (EBNF) [42] grammar shown in Fig. 3, interpreted under set semantics (no duplicates by using DISTINCT) and under binary logic (no null values allowed in the input tables). We also require the same syntactic restriction as for TRC*: *every predicate needs to be guarded* (Definition 3), i.e., every predicate must reference at least one table within the scope of the last NOT. This restriction excludes hidden disjunctions, such as "NOT(NOT(P1) and NOT(P2))" which is equivalent to "P1 or P2".

Definition 5. SQL*: Non-disjunctive SQL under set semantics (SQL*) is the syntactic restriction of SQL under binary logic (no NULL values in the input tables) to the grammar defined in Fig. 3, and additionally requiring every predicate to be guarded.

Every SQL* query can be brought into a canonical form that maintains a straightforward one-to-one correspondence with TRC*. The idea is to replace membership and quantified subqueries with existential subqueries (see grammar in Fig. 3) and then unnest any existential quantifiers, i.e., to only use "not exists". This pulling up quantification as early as possible is identical to the way we defined the canonical form of TRC*.

The Datalog$^\neg$ query from (3) cannot be expressed in SQL* and requires either a UNION operator or disjunction as in Fig. 4.

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

Fig. 3. 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 3), i.e., reference at least one table within the scope of the last NOT.

```sql
SELECT DISTINCT R.A
FROM R, S, T
WHERE R.B > 5
AND (R.A = S.A OR R.A = T.A)
```

Fig. 4. Example SQL with disjunction.

2.5 Logical expressiveness of the fragment

We show that the four languages restricted to the non-disjunctive fragment are equivalent in their logical expressiveness. The proof is available in the optional online appendix [32] and is an adaptation of the standard proof of equal expressiveness as found, for example, in [54]. However, the translations also need to pay attention to the restricted fragment (e.g. we cannot use union to define an active domain) and attempt to keep the numbers of extensional database atoms the same, if possible. This detail will be important later in Section 4, where we show that those four fragments differ in the types of query patterns they can express.

Theorem 6. *[Logical expressiveness] Datalog*, RA*, TRC*, and SQL* have the same logical expressiveness.*

3 RELATIONAL DIAGRAMS*

This section introduces our diagrammatic representation of relational queries. It details the basic visual elements of Relational Diagrams*(Section 3.1), gives the formal translation from TRC* (Section 3.2) and back (Section 3.3), and shows that there is a one-to-one correspondence between TRC* expressions and Relational Diagrams*, thereby proving their validity (Section 3.4).

3.1 Visual elements

In designing our diagrammatic representation, we started from existing widely-used visual metaphors and then added the minimum necessary visual elements to obtain expressiveness for full TRC*. In the following five points, we discuss both (*i*) necessary specifications for Relational

Diagrams* and (*ii*) concrete design choices that are not formally required but justified based on best practices from HCI and visualization guidelines. We use the term *canvas* to refer to the plane in which a Relational Diagram* is displayed. We illustrate with Fig. 5.

(1) Tables and attributes: We use the set-of-mappings definition of relations [54] in which a tuple is a mapping from attributes' names to values, in contrast to the set-of-lists representation in which order of presentation matters and which more closely matches the typical vector representation. Thus a table is represented by any visual grouping of its attributes. We use the typical UML convention of representing tables as rectangular boxes with table names on top and attribute names below in separate rows. Table names are shown with white text on a black background and, to differentiate them, attributes use black text on a white background. For example, table R with attribute $\boxed{\text{A}}$. Similar to Datalog and RA (and different from SQL and TRC), we do not use table aliases. Such table aliases create extra cognitive burden and are only needed in languages where references to repeated table instances cannot be otherwise disambiguated. We also reduce visual complexity by only showing attributes that are used in the query, similar to SQL and TRC (and different from Datalog). Database users are commonly familiar with relational schema diagrams. Thus, we believe that a simple conjunctive query should be visualized similarly to a typical database schema representation, as used prominently in standard introductory database textbooks [25, 51].

(2) Selection predicates: Selection predicates are filters and are shown "in place." For example, an attribute "$r_2.C > 1$" is shown as $\boxed{\text{C>1}}$ in the corresponding instance of table R. An attribute participating in multiple selection predicates is repeated at least as many times as there are selections (e.g. to display "$r_2.C > 1 \wedge r_2.C < 3$", we would repeat R.C twice as $\boxed{\text{C>1}}$ and $\boxed{\text{C<3}}$). An attribute participating in k selection predicates is repeated k times.

(3) Join predicates: Equi-join predicates (e.g. "$s_2.A = t_2.A$"), which arguably are the most common type of join in practice, are represented by lines connecting joined attributes. For other less-frequent theta join operators $\{\neq, <, \leq, \geq, >\}$, we add the operator as a label on the line and use an arrowhead to indicate the reading order and correct application of the operator *in the direction of the arrow*. For example, for a predicate "$r_1.A > r_2.B$", the label is > and the arrow points from attribute A of the first R occurrence to B of the second: $\text{A}\xrightarrow{>}\text{B}$. Notice that the direction of arrows can be flipped, along with flipping the operator, while maintaining the identical meaning: $\text{A}\xleftarrow{<}\text{B}$. To avoid ambiguity with the standard left-to-right reading convention for operators, we normalize arrows to never point from right to left. An attribute participating in multiple join predicates needs to be shown only once and has several lines connecting it to other attributes. An attribute participating in one or more join predicates and k selection predicates is shown $k + 1$ times.[3]

(4) Negation boxes: In TRC*, negations are either avoided (e.g. $\neg(R.A = S.B)$ is identical to $R.A \neq S.B$) or placed before the existential quantifiers. We represent a negation with a closed line that partitions the canvas into a subcanvas that is negated (inside the bounding box) and everything else that is not (outside of the bounding box). As a convention, we use dashed rounded rectangles.[4] Recursive partitioning of the canvas allows us to represent a tree-based nesting order that corresponds to the nested scopes of quantified tuple variables in TRC (and also 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]In practice, one can reduce the size of a Relational Diagram* by reusing an existing selection predicate for joins. This comes at the conceptual complication that the exact graph topology of the Relational Diagram* (which attributes are connected) is not uniquely determined (though it still allows only one correct interpretation). In our example Fig. 5d, one could remove the attribute R.C of r_2 and connect Q.D to either $\boxed{\text{C>1}}$ or $\boxed{\text{C<3}}$ instead.

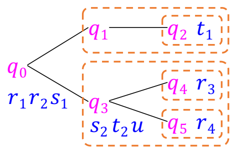
[4]Rectangles allow better use of space than ellipses, and rounded corners together with dashed lines distinguish those negation boxes clearly from the rectangles with solid edges and right angles used for tables and attributes.

$$\begin{aligned}&\{q(A,D) \mid \exists r_1{\in}R, \exists r_2{\in}R, \exists s_1{\in}S[q.A{=}r_1.A \wedge q.D{=}r_2.C \wedge \\ &\quad r_2.C{>}1 \wedge r_2.C{<}3 \wedge r_1.A{>}r_2.B \wedge \\ &\quad \neg(\neg(\exists t_1{\in}T[t_1.A{=}r_1.A])) \wedge \\ &\quad \neg(\exists s_2{\in}S, \exists t_2{\in}T, \exists u{\in}U[s_2.A{=}t_2.A \wedge s_2.B{>}s_1.A \wedge \\ &\qquad \neg(\exists r_3{\in}R[r_3.A{\neq}1]) \wedge \neg(\exists r_4{\in}R[r_4.B{\neq}s_2.B])])]\}\end{aligned}$$

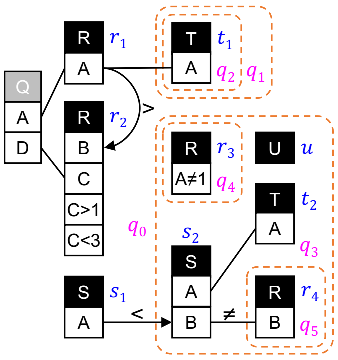
(a) TRC*

$$\begin{aligned}&q_0 \wedge \\ &\quad \neg(q_1 \wedge \\ &\qquad \neg(q_2)) \wedge \\ &\quad \neg(q_3 \wedge \\ &\qquad \neg(q_4) \wedge \\ &\qquad \neg(q_5))\end{aligned}$$

(b) Negation hierarchy



(c) Canvas partitions



(d) Relational Diagram*

$$\begin{aligned}&\exists r_1{\in}R, \exists r_2{\in}R, \exists s_1{\in}S[\ldots \wedge \\ &\quad \neg(\\ &\qquad \neg(\exists t_1{\in}T[\ldots])) \wedge \\ &\quad \neg(\exists s_2{\in}S, \exists t_2{\in}T, \exists u{\in}U[\ldots \wedge \\ &\qquad \neg(\exists r_3{\in}R[\ldots]) \wedge \\ &\qquad \neg(\exists r_4{\in}R[\ldots])])]\end{aligned}$$

(e) TRC* stub

Fig. 5. Section 3.2: Example TRC* expression (a), derivation of the negation hierarchy (b, c), and corresponding Relational Diagram* (d). Colored partitions q_i (purple) and table variables r_i (blue) are not part of the diagram and shown only to discuss the correspondence. Section 3.3: TRC* stub after step 2 of the translation (e).

(5) Output table: We display an output table to emphasize the compositional nature of relational queries: a relational query uses several tables as input, and returns one new table as output. We use the same symbol for that output table as the TRC expression, for which we most commonly use Q. We use a gray background $\boxed{Q}$ to make this table visually distinct from input tables.

3.2 From TRC* to Relational Diagrams*

We next describe the 5-step translation from any valid TRC* expression to a Relational Diagrams*. We illustrate by translating a TRC* expression (Fig. 5a) into a Relational Diagrams* (Fig. 5d). Notice that the translation critically leverages three conditions fulfilled by the input: (1) Safe TRC (and thus also TRC*) only allows existential and not universal quantification [54], (2) TRC* only allows conjunction between predicates, and (3) all predicates in TRC* are guarded (recall Definition 3).

(1) Creating canvas partitions: The scopes of the negations in a TRC are nested by definition. We translate this hierarchy of the scopes for each negation (the *negation hierarchy*) into a nested partition of the canvas. Fig. 5c illustrates the nested partitions as derived from the negation hierarchy Fig. 5b of the original TRC* expression. Notice that the double negation "$\neg(\neg(\ldots))$" results in the scope q_1 of the negation hierarchy to be empty.

(2) Placing tables: Each table variable defines a table that gets placed into the canvas partition that corresponds to the respective negation scope. For example, the tables corresponding to the table variables r_1, r_2, and s_1 are outside any negation scope and thus placed in the root partition q_0. Similar to Datalog and RA (and in contrast to TRC and SQL) Relational Diagrams* do not need table aliases.

(3) Placing selection predicates: The predicates within each scope are combined via conjunction and are thus added one after the other. Since all selection predicates are guarded, the selection predicates

can be placed in the same partition as their respective table, which allows correct interpretation (see Section 3.3). For example, for $\neg(\exists r_3 \in R[r_3.A \neq 1])$, the predicate "$A \neq 1$" is placed directly below R in q_4. An example of a predicate that is not guarded would be $\exists r_3 \in R[\neg(r_3.A = 1)]$: the scope of the negation contains a predicate of a table that is not existentially quantified in that scope.

(4) Placing join predicates: For each join predicate, we add the two attributes (if not already present) and connect them via an edge with any comparison operator drawn in the middle. An attribute participating in multiple join predicates needs to be shown only once. Equi-joins are the standard and no operator is shown. Asymmetric joins include an arrowhead at one end of the edge (see Section 3.1). As for guarded join predicates one or both attributes are in the partition of a local table, the negation can be correctly interpreted. An example of an unguarded predicate would be $\neg(r_4.B = s_2.B)$. What is possible is the logically-equivalent $r_4.B \neq s_2.B$ (as long as one of the two attributes is in the local scope of the last negation. In our example, this is the case in $\neg(\exists r_4 \in R[r_4.B \neq s_2.B])$.

(5) Place and connect output table: The safety conditions for TRC [54] imply that output predicates can only be chosen from tables outside of all negations, thus in the root scope or partition q_0. If the query is non-Boolean, we add a new table named Q for query with a unique gray background $\boxed{Q}$ to imply the difference from table references. If the query is Boolean, there is no output table and the query represents a logical sentence that is true or false.

Completeness. This five-step translation guarantees uniqueness of the following aspects: (1) nesting hierarchy (corresponding to the negation hierarchy), (2) where tables are placed (canvas partitions corresponding to the negation scope), (3) which attributes have selection predicates, and (4) which attributes participate in joins and how. The following aspects are not uniquely defined (without impact on the later interpretation): (1) the order of attributes below each table; (2) the direction of arrows can be flipped with a simultaneous label flip e.g., $s_1.A \xleftarrow{\geq} s_2.B$ and $s_1.A \xrightarrow{\leq} s_2.B$ are identical (by convention we avoid arrows from right-to-left, but allow them up-to-down and down-to-up); (3) the size of visual elements and their relative arrangement; and (4) any optional changes in style (e.g. other than dashed negation boxes, distinct visual appearance between tables and attributes).

3.3 From Relational Diagram*to TRC

We next describe the reverse five-step translation from any valid Relational Diagram* to a valid and unique TRC* expression. At the end, we summarize the conditions of a Relational Diagram* to be valid, which are the set of requirements listed for each of the five steps. We again illustrate with the examples from Fig. 5.

(1) Determine the nested scopes of negation: From the nested canvas partitions (Fig. 5c), create the nested scopes of the negation operators of the later TRC* expression (Fig. 5b).

(2) Quantification of table variables: Each table in a partition corresponds to an existentially-quantified table variable. WLOG, we use a small letter indexed by the number of occurrence for repeated tables. We add those quantified table variables in the respective scope of the negation hierarchy (Fig. 5c). For example, table T in q_2 becomes $\exists t_1 \in T[\ldots]$ and replaces q_2 in Fig. 5e. Notice that partition q_1 is empty and the resulting negation scope does not contain any expression other than another negation scope. We require that the leaves of the partition are not empty and contain at least one table. Otherwise, expressions $\wedge\neg()$ and $\wedge\neg(\neg())$ would both have to be true, leaving the meaning of an empty leaf partition ambiguous. This also implies that an empty canvas (there is only one partition, in which root and leaf are empty) is not a valid Relational Diagram*.

(3) Selection predicates: Selection attributes are placed into the scope in which its table is defined. For example, the predicate $R.A \neq 1$ in partition q_4 leads to $\neg(\exists r_3 \in R[r_3.A \neq 1])$.

(4) Join predicates: For join predicates (lines connecting attributes in Relational Diagrams* with optional direction and operator), we have a validity condition that they can only connect attributes of tables that are in the same partition or different partitions that are in a direct-descendant relationship. In our example, $T.A$ in q_2 connects to $R.A$ in q_0 (here q_0 is the root and grandparent of q_2.) However, we could not connect any attribute in q_5 with any attribute in q_4 (which are siblings in the nesting hierarchy). This requirement is the topological equivalent of scopes for quantified variables in TRC and guarantees that only already-defined table variables are referenced. Each such predicate is placed in the scope of the lower of the two partitions in the hierarchy, which guarantees the predicate to be guarded. For example, the inequality join connecting $S.B$ in q_3 and $R.B$ in q_5 is placed in the scope of q_5.

(5) Output table: The validity condition for the output table is that each of its one or more attributes is connected to exactly one attribute from a table in the root partition q_0. This corresponds to the standard safety condition of safe TRC. This step adds the set parentheses, the output tables, and its attribute and output predicates shown in green in Fig. 5a for non-Boolean queries.

Soundness. Notice that this five-step translation guarantees that the resulting TRC* is uniquely determined up to (1) renaming of the tuple variables; (2) reordering the predicates in conjunctions, and (3) flipping the left/right positions of attributes in each predicate. It follows that Relational Diagrams* are sound, and their logical interpretation is *unambiguous*.

3.4 Valid Relational Diagrams*

In order for a Relational Diagrams*to be valid we require that each of the conditions for the five-step translation process is fulfilled.

Definition 7 (Validity). A Relational Diagram* is valid iff:

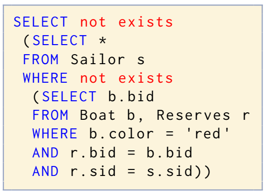
(1) The nested hierarchy of optional negation boxes partitions the canvas (any two dashed boxes are either disjoint or one is completely contained within the other).
(2) Each table, its attributes, and its selection predicates are discernible and reside in exactly one canvas partition.
(3) Each leaf in the canvas partition contains at least one table.
(4) Joins only happen between attributes of tables in partitions that are descendants (not siblings or their descendants). Join predicates with asymmetric operators such as < and > require a line with directionality (e.g. an arrowhead).
(5) If there is an output table, then it has at least one attribute, and each attribute connects to exactly one attribute in the root partition q_0 (safety condition of TRC).

Theorem 8 (Unambiguous Relational Diagrams*). *Every valid Relational Diagram* has an unambiguous interpretation in TRC*.*

The constructive translations from Sections 3.2 and 3.3 form the proof. Also notice that there is an additional validity condition that we will add later in Definition 16 that will extend the logical expressiveness to include disjunction and go beyond TRC*.

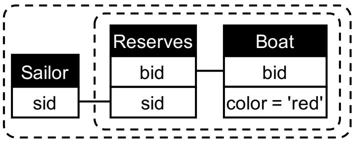
3.5 Logical statements

Boolean queries (or logical sentences) are formulas without free variables. Being able to express *relational sentences* (or constraints) allows us to compare our formalism against a long history of formalisms for logical statements [31]. An additional freedom with sentences is that the otherwise important safety conditions of relational calculus vanish. Thus, we need to be able to express statements that do not have any existentially-quantified relations in the main canvas. We next give an intuitive example, with more examples given in the online appendix [32].



```
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)

Fig. 6. Example 3: All sailors reserve some red boat.

Example 3. *Consider the statement:* "All sailors reserve a red boat."

$$\neg(\exists s \in Sailor[\neg(\exists b \in Boat, r \in Reserves[b.color = 'red' \wedge r.bid = b.bid \wedge r.sid = s.sid])]) \tag{4}$$

The first 4 steps of the translation in Section 3.2 still work: the root canvas q_0 does not contain any relation (Fig. 6b). Similarly, the equivalent canonical SQL statement contains no FROM clause before the first NOT. Notice that Definition 12 of query pattern isomorphism still works as it is defined based on the relational tables.*

3.6 A note on implementation

Creating valid Relational Diagrams* programmatically requires a spatial layout algorithm that ensures that tables, predicates, and nested multi-layer canvas partitions are drawn unambiguously. To improve readability, it should also reduce edge crossings and edge bendiness. For initial work in that direction, please see our optimization model approach called STRATISFIMAL LAYOUT [21].

4 RELATIONAL QUERY PATTERNS

Example 1 illustrated that—while two languages may well have the same logical expressiveness—one of them may have more ways to represent "logical patterns" than the other. We are interested in making this intuition more precise and *establishing a language-independent formalism* that captures the so-far vague notion of a relational query pattern. The formalism should allow us to study the "relative pattern expressiveness" of languages, i.e.: *Can languages $\mathcal{L}_2$ express all patterns that language $\mathcal{L}_1$ can?* We will then apply this formalism to the non-disjunctive fragment and compare the four previously-defined relational query languages by their relative abilities to represent "the same set of patterns" as other languages.

In the following, we often need to distinguish between a query as the *query expression* (the actual syntax in a particular query language) and a query as a *logical function* that maps a set of input tables to an output table. If we need to be precise, we refer to the function implied by a query as the *query semantics* and the actual syntax as the *query expression*. We use the word *signature* to refer to an ordered argument list as the input to a function, and use bracket notation for indexing. For example, the signature of $f(x, y)$ is $\mathcal{S} = (x, y)$, and the first element is $\mathcal{S}[1] = x$.

4.1 Defining relational query patterns

Intuition. Our goal is to define relational patterns in a way that allows us to analyze and compare *any relational query languages* irrespective of their syntax. Our idea is to formalize patterns based on the only common symbols in queries across languages: references to the *input relations* from the database. Since every relational query language needs to use input tables, the resulting formalism generalizes. Intuitively, we will define two queries to be *pattern-isomorphic*[5] 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 input tables (e.g. inserting a tuple), both queries will make identical changes to their outputs. However, for queries with multiple occurrences of the same input table (also called self-joins), we need to treat such repeated occurrences of the same input table as if they were *independent tables*. We will refer to such repeated table occurrences as "*table references*."

For example, consider $q = R - (\pi_A R \times S)$. The semantics of this query expression is a function $q(R, S)$ that maps input relations R and S to an output table, and its signature would be just its relational input (R, S). However, we will not be interested in the signature of a query semantics, but rather the *signature of a query expression*, since we need to capture that two occurrences of R play different semantic roles in the query. In order to capture these different roles, we assign unique names to each table reference, resulting in what we call the *dissociated query* $q' = R_1 - (\pi_A R_2 \times S)$. We then formally define the *relational query pattern* of q as the logical function $q'(R_1, R_2, S)$ expressed by the dissociated query q'. Notice that our definition of "dissociation" is inspired by, yet slightly different from its original use in the context of probabilistic inference [34] and the complexity of resilience and causal responsibility [26].

Formalization. To make our definitions precise across relational query languages with different syntax, we need to unambiguously refer to the individual occurrences of relational input tables in a given query expression, irrespective of the language used.

Definition 9 (Query signature). A *table reference* in a query expression q is any existentially or universally quantified reference to an input table. The *signature* $\mathcal{S}$ of q is the ordered list of its table references.

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 "$R(x, \_)$".[6] In contrast, the symbol "R" is not a table reference in the SQL fragment "WHERE R=1" as it is part of a reference to an attribute of a previously-defined table variable and not part of an existentially-quantified statement. The signature of a conjunctive SQL query with FROM clause "FROM R as R1, R as R2, S" is $\mathcal{S} = (R, R, S)$.

Definition 10 (Dissociated query). A dissociation of a query expression q with signature $\mathcal{S}$ is a modified query q' with $\mathcal{S}$ being replaced with a table signature $\mathcal{S}'$ of same size (i.e. $|\mathcal{S}'| = |\mathcal{S}|$), where every table in $\mathcal{S}'$ has a different name, and every table $\mathcal{S}'[i]$ has the same schema as table $\mathcal{S}[i]$ for all $i \in [|\mathcal{S}|]$.

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

Example 4 (Dissociation). *The RA query $q = R - (\pi_A R \times S)$ has signature $\mathcal{S} = (R, R, S)$ with two of the three table references referring to the same input table R. Replacing the signature $\mathcal{S}$ with a dissociated signature $\mathcal{S}' = (R_1, R_2, S)$ leads to the dissociated query $q' = R_1 - (\pi_A R_2 \times S)$. Since the dissociated tables R_1, R_2 inherit the schema information from table R, the dissociated query is still a*

[5]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) [27]. We use the term *pattern-isomorphic* (instead of structure-isomorphic) since our focus is on particular relational structures we refer to as patterns.
[6]Existential quantification either happens explicitly as in TRC, or implicitly as in RA.

valid query and represents a new relational function $q'(R_1, R_2, S)$ that maps three different *input tables to an output.*

The intuition behind this formalism is that the dissociated query defines a function that maps a set of *table references* (not just a set of input tables) to an output table. Thus, *the dissociated query is a semantic definition* of a relational query pattern across different relational query languages. Two queries use the same query pattern if their dissociated queries are logically equivalent, up to renaming and reordering of the input tables.

Definition 11 (Relational pattern). Given a query expression q with signature $\mathcal{S}$. The *relational pattern* of q is the logical function defined by its dissociated query $q'(\mathcal{S}')$.

Definition 12 (Pattern isomorphism). Given two logically-equivalent queries q_1 and q_2 with signatures $\mathcal{S}_1$ and $\mathcal{S}_2$, and dissociated queries $q_1'(\mathcal{S}_1')$ and $q_2'(\mathcal{S}_2')$, respectively. The queries are *pattern-isomorphic* iff $q_1'(\mathcal{S}_1') = q_2'(\pi(\mathcal{S}_1'))$ for some permutation π. In that case, we call the bijection $\mathcal{S}_1[i] \mapsto \mathcal{S}_2[\pi(i)]$ between the query signatures a *pattern-preserving mapping*.

Example 5 (Patterns). *Next, consider the Datalog query*

$$I(x, y) :\!- R(x, \_), S(y).$$
$$Q_1(x, y) :\!- R(x, z), \neg I(x, y).$$

with signature $\mathcal{S}_1 = (R, S, R)$. Then its dissociated query is

$$I(x, y) :\!- R_1(x, \_), S_2(y).$$
$$Q'(x, y) :\!- R_3(x, z), \neg I(x, y).$$

with signature $\mathcal{S}_1' = (R_1, S_2, R_3)$. Notice that Q' defines a logical function $Q'(R_1, S_2, R_3)$ mapping two input tables with the same schema as R and an input table S to a binary output table.

Next, consider the RA query q from Example 4 with signature $\mathcal{S}_2 = (R, R, S)$. Notice that above Datalog query Q and this RA query q are pattern-isomorphic since their dissociated queries define the same logical function up to permutation in the signatures: $Q_1'(R_1, S_2, R_3) = q'(R_3, R_1, S_2) = q'(\pi(R_1, S_2, R_3))$ for permutation $\pi = (2, 3, 1)$. Thus the mapping $(\mathcal{S}_1[1], \mathcal{S}_1[2], \mathcal{S}_1[3]) \mapsto (\mathcal{S}_2[2], \mathcal{S}_2[3], \mathcal{S}_2[1])$ is a pattern-preserving mapping.

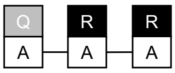
Complexity of deciding pattern isomorphism. Deciding whether two relational queries are pattern-isomorphic is undecidable, in general (we need to determine whether two queries are equivalent, both before and after dissociation). This follows from Trakhtenbrot's theorem stating that the problem of validity in first-order logic on finite models is undecidable, and thus also the logical equivalence of relational queries (see, e.g., the reduction in [6]). However, we get a one-sided guarantee: if we can determine whether two queries are logically equivalent, then we can also determine whether they are pattern-isomorphic. In practice, equivalence of relational queries can often be determined, even for sophisticated SQL queries with grouping and aggregation evaluated over bags or sets [13].

4.2 Discussion with illustrating example

We next give an example of queries that have different query patterns *although they are logically equivalent and have the same table signature.* This detailed example motivates to a large extent why we define query patterns based on the *dissociated* signature.

```
SELECT DISTINCT R1.A
FROM R R1, R R2
WHERE R1.A = R2.A
```

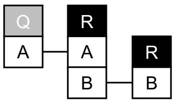
(a)



(b)

```
SELECT DISTINCT R1.A
FROM R R1, R R2
WHERE R1.B = R2.B
```

(c)



(d)

Fig. 7. Example 6: Two queries (a) and (c) with identical signatures that are logically equivalent but *not pattern-isomorphic*. Their associated Relational Diagrams are shown in (b) and (d), respectively.

Example 6 (Different patterns). *Consider table $R(A, B)$ and the two Datalog queries $Q_1(R)$ and $Q_2(R)$ with*

$$Q_1(x) :\!- R(x, \_), R(x, \_).$$
$$Q_2(x) :\!- R(x, y), R(\_, y).$$

Both queries are logically equivalent to $Q(x) :\!- R(x, \_)$, and thus also logically equivalent to each other. However, Q_1 and Q_2 represent different patterns: Q_1 never uses the second attribute of R whereas Q_2 uses that attribute to join both occurrences of R. This difference becomes even more apparent in SQL: Fig. 7a would even work if R was unary, whereas Fig. 7c requires R to be at least binary.

We next show that table dissociation allows us to formally distinguish the two patterns. First, notice that both queries have two occurrences of R as table references, and hence we need to associate each individual appearance to the "role" it plays semantically in the query. We achieve this by first dissociating the two occurrences of R into two fresh input tables (with the same schema). The two dissociated queries are $Q_1'(R_1, R_2)$ and $Q_2'(R_3, R_4)$ with

$$Q_1'(x) :\!- R_1(x, \_), R_2(x, \_).$$
$$Q_2'(x) :\!- R_3(x, y), R_4(\_, y).$$

It is easy to verify that neither of the two possible mappings $h_1 = \{(R_1, R_3), (R_2, R_4)\}$ and $h_2 = \{(R_1, R_4), (R_2, R_3)\}$, preserves logical equivalence for the dissociated queries. For example, $R_1(1, 2), R_2(1, 3)$ returns an answer for Q_1' but not for Q_2', under neither h_1 nor h_2.

However, Q_1 is pattern-isomorphic to the TRC query $q_3(R)$ with

$$\{q_3(A) \mid \exists r_1 \in R, r_2 \in R[q.A = r_1.A \wedge r_1.A = r_2.A]\}$$

To see that, notice that its dissociated query $q_3'(R_5, R_6)$ with

$$\{q_3'(A) \mid \exists r_1 \in R_5, r_2 \in R_6[q.A = r_1.A \wedge r_1.A = r_2.A]\}$$

allows the isomorphism $h_3 = \{(R_1, R_5), (R_2, R_6)\}$ from Q_1' to q_3' that preserves logical equivalence. By the same arguments, Q_1 is pattern-isomorphic to the SQL query in Fig. 7a, and Q_2 is pattern-isomorphic to the SQL query in Fig. 7c.

By design, our definition excludes views and intermediate tables such as Intensional Database Predicates in Datalog from the definition of table references. To see why, consider a query returning nodes that form the starting point of a length-3 directed path:

$$Q_1(x) :\!- E(x, y), E(y, z), E(z, w).$$

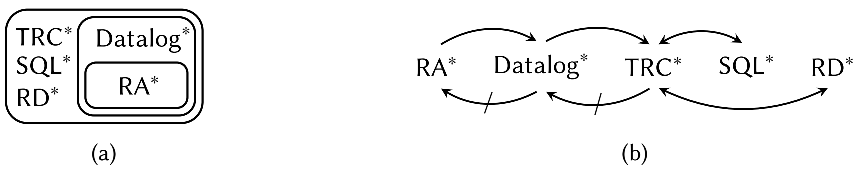


Fig. 8. Theorem 14: (a) A diagram summarizing the representation hierarchy between the non-disjunctive fragments of 4 query languages and Relational Diagrams* (shown as RD*). (b) Directions of pattern-preservation (and non-preservation) used in the proofs.

The following Datalog query uses the same logical pattern (find three edges that join and keep the starting node), even though it defines the intermediate intensional database predicate I:

$$I(y) :- E(y,z), E(z,w).$$
$$Q_1'(x) :- E(x,y), I(y).$$

4.3 Comparing the “pattern-expressiveness” of relational languages

We next add the final definition needed to formally compare relational query languages based on their relative abilities to represent relational query patterns.

Definition 13 (Representation equivalence). We say that a query language $\mathcal{L}_2$ can *pattern-represent* a query language $\mathcal{L}_1$ (written as $\mathcal{L}_1 \subseteq^{\text{rep}} \mathcal{L}_2$) iff for every legal query expression $q_1 \in \mathcal{L}_1$ there is a pattern-isomorphic query $q_2 \in \mathcal{L}_2$. We call a query languages $\mathcal{L}_2$ *pattern-dominating* another language $\mathcal{L}_1$ (written as $\mathcal{L}_1 \subsetneq^{\text{rep}} \mathcal{L}_2$) iff $\mathcal{L}_1 \subseteq^{\text{rep}} \mathcal{L}_2$ but $\mathcal{L}_1 \not\supseteq^{\text{rep}} \mathcal{L}_2$. We call $\mathcal{L}_1, \mathcal{L}_2$ *representation equivalent* (written as $\mathcal{L}_1 \equiv^{\text{rep}} \mathcal{L}_2$) iff $\mathcal{L}_1 \subseteq^{\text{rep}} \mathcal{L}_2$ and $\mathcal{L}_1 \supseteq^{\text{rep}} \mathcal{L}_2$, i.e., both language can represent the same set of relational patterns.

We are now ready to state our result on the hierarchy of pattern expressiveness of the non-disjunctive fragment of the four languages defined earlier (Section 2) and our proposed relational diagrammatic representation Relational Diagrams* (Section 3):

Theorem 14 (Representation hierarchy). *RA** $\subsetneq^{rep}$ *Datalog** $\subsetneq^{rep}$ *TRC** $\equiv^{rep}$ *SQL** $\equiv^{rep}$ *Relational Diagrams* (see Fig. 8).*

In addition to containing all proofs, our optional online appendix [32] shows how the separation between RA* and Datalog* disappears after adding the antijoin operator to the basic operators of RA*, while the separation from TRC* remains. The proof demonstrates that *relational calculus has relational patterns that cannot be expressed in relational algebra.* The important consequence is that *RA*, Datalog* or any diagrammatic language modeled after them would not be a suitable target language for helping users understand all existing relational query patterns* (including those used by SQL*). Our related work (Section 7) shows that most existing visual query representations are modeled after relational algebra in that they model data flowing between relational operators, which implies they cannot faithfully represent all relational query patterns from TRC* or SQL*.

4.4 Similar patterns across schemas

We next extend the notion of pattern equivalence to allow comparing queries across *different schemas*. We call this concept “*pattern similarity*” and define it as a Boolean condition: two queries either have a similar pattern or not. The intuition is simple and best illustrated with the two queries from Fig. 2: As written those queries are not logically equivalent and thus they can't be pattern-isomorphic. However, if we first replace the table and attribute names from q_1 with table names from q_2 in a reversible (thus bijective) way, then the thus modified query q_1' would be pattern-isomorphic to q_2.

More formally, call a *schema mapping* λ from query q_1 to q_2, a bijective mapping that replaces table names, attribute names, constants, and attribute order appearing in q_1 with those from q_2.

Definition 15 (Similar Patterns). Given two queries q_1 and q_2. The queries use a *similar pattern* iff there is a schema mapping λ from q_1 to q_2 s.t. $\lambda(q_1)$ and q_2 are pattern-isomorphic.

Example 7. *For our example from Fig. 2, consider a mapping λ that replaces 'Sailor' with 'SX', 'Reserves' with 'SPX', 'Boat' with 'PX', 'sname' with 'sname', 'sid' with 'sno', and 'bid' with 'pno'. Then the thus modified query $\lambda(q_1)$ is pattern-isomorphic with q_2.*

5 RELATIONAL COMPLETENESS

To make Relational Diagrams relationally complete, we now remove the non-disjunction restrictions and allow disjunctions and unions in all four relational query languages (Section 2). This means we must also add a corresponding syntactic device to Relational Diagrams that achieves logical equivalence to the other relational query languages. Unfortunately, this means that Relational Diagrams are no longer representation-equivalent to TRC. Can this be addressed in the future by a better diagram design? Based on the current understanding of the inherent limits of diagrams to express disjunctive information [49, 50] (see also the colored car example in the online appendix [32]), such an extension would require adding *non-diagrammatic abstractions* (also called "syntactic devices").

The syntactic device that makes Relational Diagrams relationally complete is inspired by the representation of disjunction in Datalog. It was also proposed by Peirce in his discussion of Euler diagrams [43, 4.366] (see also [49, sect. 2.3.1]): we allow placing several Relational Diagrams* on the same canvas, each in a separate *union cell*. Each cell of the canvas then represents only conjunctive information, yet the relation among the different cells is disjunctive (a union of the outputs).

We next illustrate with two examples logical transformations that are not pattern-preserving but that guarantee relational completeness. These transformations, together with union cells, make Relational Diagrams *relationally complete*: every query expressible in full RA, safe TRC, Datalog$^\neg$, 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 8 (Replacing disjunctions). *Consider the SQL query 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, yet leads to an increased number of table references:*

$$\begin{aligned}
&\{q(A) \mid \exists r \in R[q.A = r.A \wedge \neg(\exists s \in S\\
&\qquad [\neg(\exists r_2 \in R[(r_2.B = s.B \vee r_2.C = s.C) \wedge r_2.A = r.A])])]\}\\
=&\{q(A) \mid \exists r \in R[q.A = r.A \wedge \neg(\exists s \in S\\
&\qquad [\neg(\exists r_2 \in R[r_2.B = s.B \wedge r_2.A = r.A]) \wedge\\
&\qquad \neg(\exists r_3 \in R[r_3.C = s.C \wedge r_3.A = r.A])]]\}
\end{aligned}$$

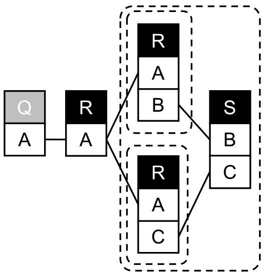
[7]Extend the grammar from Fig. 3 with one additional rule: P::= '('P OR P')' and making adjustments for allowing the UNION clause between non-Boolean queries.

```
SELECT DISTINCT R.A
FROM R
WHERE not exists
 (SELECT *
 FROM S
 WHERE not exists
  (SELECT *
  FROM R AS R2
  WHERE (R2.B = S.B
  OR R2.C = S.C)
  AND R2.A = R.A))
```

(a)

```
SELECT DISTINCT R.A
FROM R
WHERE not exists
 (SELECT *
 FROM S
 WHERE not exists
  (SELECT *
  FROM R AS R2
  WHERE R2.B = S.B)
  AND R2.A = R.A)
 AND not exists
  (SELECT *
  FROM R AS R2
  WHERE R2.C = S.C
  AND R2.A = R.A))
```

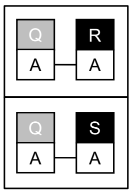
(b)



(c)

```
(SELECT
 DISTINCT R.A
FROM R)
UNION
(SELECT
 DISTINCT S.A
FROM S)
```

(d)



(e)

Fig. 9. Illustrations for Example 8 on replacing disjunctions: (a) SQL with disjunctions, (b) logically-equivalent (yet not representation-equivalent) SQL* statement, and (c) Relational Diagrams. Illustrations for Example 9 on creating the union of queries: (d) union of SQL* statements, and (e) Relational Diagrams with union cells.

Fig. 9b shows this query as representation-equivalent SQL query, and Fig. 9c as Relational Diagram.*

Example 9 (Union of queries). *Consider two unary tables $R(A)$ and $S(A)$ and the TRC query*

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

We can write this query as a union of disjunction-free TRC queries:*

$$\{q(A) \mid \exists r \in R[q.A = r.A]\} \cup \{q(A) \mid \exists s \in S[q.A = s.A]\}$$

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

The additional validity criterion for multiple union cells follows the conditions of union or disjunction in the *named perspective* [1] of query languages: for disjunction in TRC, each operand needs to have the same arity, and the mapping between them is achieved by reusing the same variables.

Definition 16 (Validity—extending Definition 7)).

(6) The output tables in multiple cells for the same query need to have the same name and same set of attributes.

Theorem 17 (Completeness). *Relational Diagrams (Relational Diagrams* extended with union cells) are relationally complete.*

The proof is in the optional online appendix [32]. It uses the earlier proven logical expressiveness of Relational Diagrams* and the fact that disjunctions can either be rewritten with DeMorgan or be pushed to the root. It also immediately follows that Relational Diagrams* (without union cells) can already express any logical statement.

corollary 18 (Completeness). *Any logical statement in first-order logic can be expressed by a logically-equivalent Relational Diagrams*.*

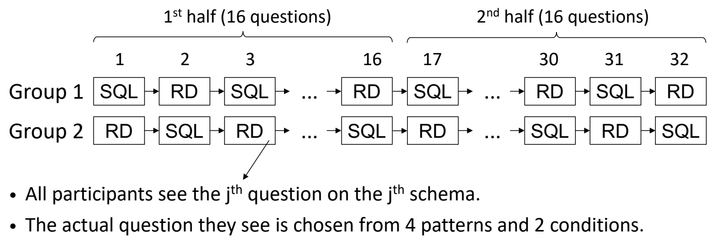

1st half (16 questions)
2nd half (16 questions)
1 2 3 16 17 30 31 32
Group 1 SQL RD SQL ... RD SQL ... RD SQL RD
Group 2 RD SQL RD ... SQL RD ... SQL RD SQL
• All participants see the jth question on the jth schema.
• The actual question they see is chosen from 4 patterns and 2 conditions.

Fig. 11. Illustration of the randomization and counterbalancing in our within-subjects study design.

6 TWO APPLICABILITY STUDIES

6.1 Relational Diagrams for Textbook Queries

We analyzed the proportion of relational calculus queries encountered in learning scenarios that have pattern-isomorphic representations in either Relational Diagrams, RA, Datalog, QueryVis [18], or QBE [57]. For that purpose, we identify 59 queries across 5 popular textbooks with sections on relational calculus [15, 19, 25, 44, 51].

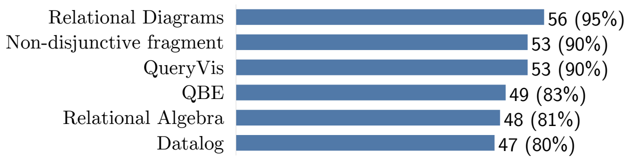


Fig. 10. Section 6.1: Fraction among 59 queries from 5 textbooks with pattern-isomorphic representations in listed languages.

Among those 59 queries, the number of queries that have pattern-isomorphic representations are 56 (94.9%) for Relational Diagrams, 53 (89.8%) for QueryVis, 49 (87.5%) for QBE, 48 (85.7%) for RA, and 47 (79.7%) for Datalog. The fraction for QueryVis happens to be identical to the *non-disjunctive fragment*. Standard Datalog cannot express disjunctions in the body of a query and thus performs worse than RA.[8] For QBE, notice that QBE 1) can express disjunctions within the same relations, yet 2) also requires the same safety conditions as Datalog. Furthermore, theta joins require the use of a non-diagrammatic conditions box [25, Appendix C]. RA extended with a primitive antijoin operator covers the same fraction as QBE. More details and all queries are given in the online appendix [32].

6.2 Controlled user study

We conducted a controlled experiment on Amazon Mechanical Turk (MTurk) [5] to evaluate the utility of Relational Diagrams for recognizing patterns. Our study investigates 3 main questions: (1) Can SQL users *identify common relational query patterns faster* using Relational Diagrams than SQL? (2) *Can participants identify patterns faster over time*, thus can users learn the patterns under repeated exposure to the same patterns? (3) Do participants have a similar *accuracy* (i.e. a comparable numbers of correct responses) using Relational Diagrams or SQL? We chose SQL as a baseline for comparison because we expect that fewer workers on MTurk understand TRC.

Open practices. Following best practices in user design, we preregistered the study design on OSF before collecting the data [32]. All code for generating the stimuli, the stimuli, the tutorial provided to participants, the resulting data (pilot $n = 13$, study $n = 50$), the analysis code, and changes from the preregistration are available on OSF [32]. More details on the study design and procedure are provided on arXiv [32].

[8]Modern variants of Datalog exist that can express disjunctions in the body, such as Souffle [52], but those are not standard.

Counterbalanced within-subjects study with randomization. We asked participants 32 questions, each having them identify which of 4 relational query patterns was presented. The exposure for each participant alternates between two conditions (Relational Diagrams and formatted SQL text). Each question uses a different relational schema found or adapted from common textbooks. We used counterbalancing and randomization to reduce ordering effects. Concretely, we start half the participants using SQL (group 1) and the rest using Relational Diagrams (group 2), after which participants alternate conditions with each question (see Figure 11). Moreover, we randomize the order in which patterns are presented such that each pattern-condition combination appeared twice in the first 16 questions (1^{st} half) and twice in the second 16 questions (2^{nd} half). The result is that each participant sees each of 4 patterns, in each of 2 conditions, exactly 2 times ($16 = 4 \times 2 \times 2$), in each of the 2 halves (first and second 16 questions). This randomization helps reduce cheating as well as order effects. This setup necessitated creating 256 different stimuli (32 schemas $\times$ 4 patterns $\times$ 2 conditions), creation of which we semi-automated. The 4 patterns, illustrated with the sailors-reserve-boats schema, were:

(1) Find *sailors* who have *reserved* **some** *boat.*
(2) Find *sailors* who have **not** *reserved* **any** *boat.*
(3) Find *sailors* who have **not** *reserved* **all** *boats.*
(4) Find *sailors* who have *reserved* **all** *boats.*

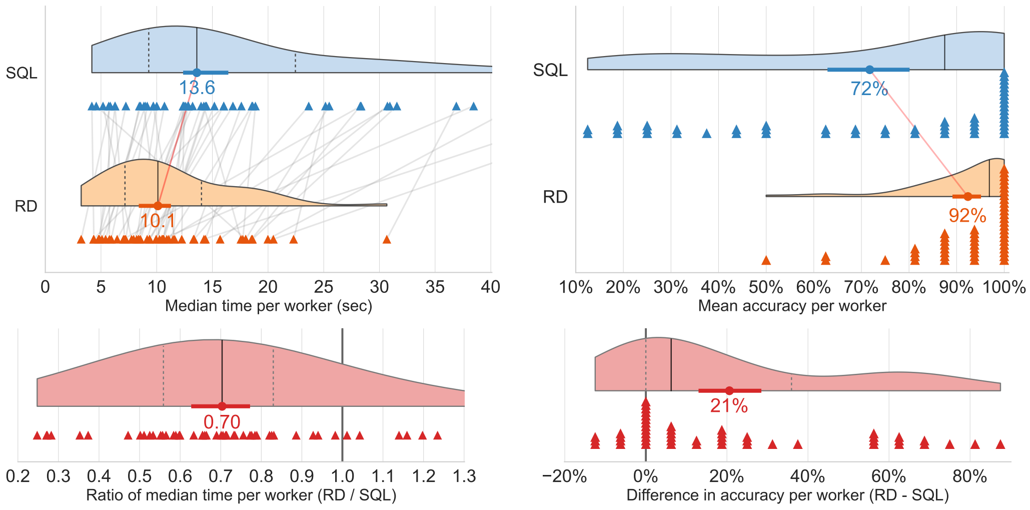
We chose these patterns because we are interested in how Relational Diagrams can be used in educational settings, and they represent 4 different query structures. In particular, double-negation from pattern (4) is challenging for novice users to understand and is easy to misinterpret [40]. Moreover, pattern (4) does not have a pattern-isomorphic representation in RA.

Several prior user studies [39, 45, 46] have shown that diagrammatic representations of queries can help users understand queries faster. Key innovations in our design are: (1) We repeatedly expose participants to 4 identical patterns across 32 different schemas, allowing us to semi-automatically design 256 questions instead of a few hand-curated ones (each participant saw only 32, one for each schema). (2) Our questions are balanced across the first 16 and second 16 questions, which allows us to track learning over time. We are not aware of prior study design that allowed studying learning in an online study. (3) Our setup is randomized and parameterized, which creates a space of $2 \cdot 2540^4$ possible treatments, i.e., participants are unlikely to see the same question sequence, reducing the chance of cheating. (4) We used monetary incentives for both time and accuracy, inspired by our recent work on QueryVis [39],

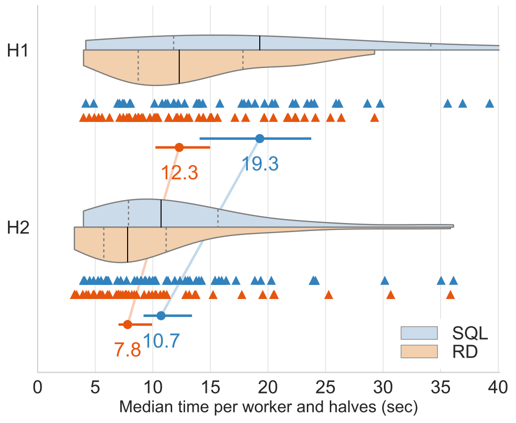
Participants. We conducted an $n = 13$ pilot study in the lab. After registering our study on OSF [32] and receiving approval from our Institutional Review Board (IRB), we began recruiting participants on MTurk [5]. Participants needed to have at least 500 completed tasks approved by requesters and at least 97% of their completed tasks approved. For us to approve their task, they needed to have at least 50% accuracy (i.e. answer at least 16 of our 32 questions correctly). Thus a participant who answered every question in SQL correctly and every question in Relational Diagrams incorrectly (or v.v.) would be included. Among the 120 task submissions, only 58 were approved. Our preregistration specified 50 participants, so we dropped 8 records and kept the counterbalancing intact by using the data from the first 25 participants who started in each condition.

Tutorial. Participants were given a self-paced 8-page tutorial on Relational Diagrams. The tutorial introduced our basic visual notations by showing SQL examples and their diagrams. The mean (resp. median) time spent on the tutorial and consent form was approximately 6.33 (respectively 3.5) minutes. The tutorial is available in our supplemental material [32].

Analysis. (1) As a within-subjects study, we first determined the *per-participant median time* for each condition. We used the median (instead of mean) for time because it is robust to outliers,



(a) Result 1. (Speed) (b) Result 3. (Accuracy)



(c) Result 2. (Learning)

Fig. 12. User study: Triangles show median times per condition or mean accuracy for each of the $n = 50$ successful participants. Violin plots show the data distribution, the median with a solid line, and the 25% and 75% quantiles with dashed lines. Error bars show the 95% BCa bootstrapped confidence intervals (CI) around the mean or median. Lines connect related marks. Relational Diagram is abbreviated here by RD.

although the median often requires more participants for the same statistical power. We computed the relative time Relational Diagrams/SQL needed per participant and again calculated the *median across* all participants. Here we used the median of the ratios as the median creates an unbiased estimator (in contrast to the mean of ratios, see the online appendix [32] for details). (2) Likewise, we computed the median per-participant time for each condition spent on the 1st half (16 questions), on the 2nd half (16 questions), and the median ratio of the 2nd/1st times. (3) We also computed the per-participant accuracy for each condition and their difference. Then, across all participants, we calculated the mean of the differences in accuracy. Here we used the mean (instead of median) since the values are bounded within [0, 1] (i.e. there are no outliers) and mean is more appropriate for discrete values like accuracies (i.e. 16/16, 15/16, ...). We analyzed these mean/median effect sizes [17, 23] and used bias-corrected and accelerated (BCa) 95% confidence intervals (CIs) to show their range of plausible values [22, 24].

Results. We summarize 3 key takeaways. The executed Python notebook has more details [32].

Result 1. *(**Speed**) We have strong evidence that participants were meaningfully faster at identifying patterns using Relational Diagrams than SQL: median ratio Relational Diagrams/SQL = 0.70, 95% CI [0.63, 0.77].*

Our choice of visualization is a variant of Raincloud plots [4] and is inspired from recent work in the visualization literature [16] discussing various ways to juxtapose multiple visualizations ("clouds + rain + lightning") in the same chart for increasing information content. In that framework, each of our charts consists of (*i*) *density plots* that show an overview of the shape of the distribution (the "cloud"), (*ii*) unjittered *dot plots* that show the raw data (the "rain": here we deviate from [16] in using triangles instead of circles which, in our opinion, are more easily countable due to their visible vertices), and (*iii*) *95% confidence intervals* that provide summary statistics (the "lightning"). Furthermore, whenever we compare alternative modalities ("repeated measures"), we also use (4) *paired plots* with lines connecting summary statistics and/or raw data.

Figure 12a (top) uses a paired plot to show the individual median per-participant times (and overall median across participants together with confidence interval) for both SQL (13.61, 95% CI [12.37, 16.43] in blue on the top) and Relational Diagrams (10.11 95% CI [8.38, 11.26] in orange on the bottom). Fig. 12a (bottom) shows the per-participant ratios between median times. Notice that the 95% confidence interval of the overall median [0.63, 0.77] does not overlap 1.00, which gives strong evidence for our conclusions.

Result 2. *(**Learning**) Participants got meaningfully faster during the study in both conditions.*

Figure 12c shows the individual times for H1 (1st half, i.e. the first 16 questions) and H2 (2nd half, i.e. the last 16 questions), for both SQL (in blue on the top) and Relational Diagrams (in orange on the bottom), together with medians and CIs. We see that the overall trend (Relational Diagrams being faster than SQL) is repeated across both halfs, and additionally that learning is taking place (participants need less time in H2 than in H1 in both conditions). The median ratios H1/H2 we used for inference (not shown but in our supplemental material) are 0.71, 95% CI [0.63, 0.79] for Relational Diagrams, and 0.70, 95% CI [0.51, 0.79] for SQL.

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%].*

Figure 12b (top) shows that the per-participant accuracies and the overall mean accuracies were meaningfully higher with Relational Diagrams than with SQL. Notice that each participant answered 16 questions in each condition, thus possible scores are $16/16 = 1$, $15/16 \approx 0.94$, etc. Thus accuracy per user and modality is discretized in multiples of $1/16$ (in contrast to completion time, which is a continuous value and differs, even if slightly, between any two users). We thus use stacked triangles akin to a Wilkinson dot plot [56] to avoid overplotting and show individual data points. Figure 12b (bottom) shows the per-participant difference in mean accuracy. As the 95% CI of the overall mean [13%, 29%] does not overlap 0, we have strong evidence for our conclusion.

Participant comments. Participants could optionally write feedback at the end of the study. Few participants did, but those who did were encouraging, such as, "I found your diagrams very helpful in understanding the queries. At first I didn't get it, but after staring at the diagrams for a few minutes it clicked and everything became super simple. I saw the patterns and it became just looking for the correct pattern to know which query was being used."

7 RELATED WORK

7.1 Peirce's beta Existential Graphs

Relational Diagrams represent nested quantifiers similarly as the influential and widely-studied *Existential Graphs* by Charles Sanders Peirce [43, 48, 50] for expressing logical statements (i.e. Boolean queries). Peirce's graphs come in two variants called alpha and beta. Alpha graphs represent propositional logic, and beta graphs represent first-order logic (FOL). Both variants use so-called *cuts* to express negation (similar to our negation boxes), and beta graphs use a syntactical element called the *Line of Identity* (LI) to denote both *the existence of objects* and *the identity between objects.*

Differences. The four key differences of beta graphs vs. Relational Diagrams are: (1) beta graphs can only represent sentences and not queries; (2) beta graphs cannot represent constants, thus selections cannot be modeled and instead require dedicated predicates; (3) beta graphs can only represent identity predicates (and no comparisons); and (4) Lines of Identity (LIs) in beta graphs have multiple meanings (existential quantification and identity between objects) and are a primary symbol.[9] This *function overload of LIs* can make reading the graphs ambiguous. We, in contrast, have predicates inspired by TRC. Lines only connect two attributes and have no loose ends. Interpreting a graph as a TRC formula is straightforward and can be summarized in a simple set of rules (recall Section 3). We discuss this important conceptual difference in detail in the online appendix [32].

7.2 QueryVis

Some of our design decisions are similar to an earlier query representation called QueryVis [18, 29, 39]. In QueryVis diagrams, grouping boxes are used to group all tables within a local scope, i.e., *for each individual query block.* Those boxes thus cannot show their respective nesting, and an additional symbol of directed arrows is needed to "encode" the nesting. The high-level consequence of those design decisions is that (1) QueryVis does not guarantee to unambiguously visualize nested queries with nesting depth ≥ 4 (please see our online appendix [32] for a minimum example), (2) each grouping box needs to contain at least one relation (thus QueryVis cannot represent the query in Fig. 5), and (3) QueryVis cannot represent general Boolean sentences (e.g. the sentence "All sailors have reserved some red boat"). Thus QueryVis is not sound and not relationally complete, even for the disjunctive fragment.

7.3 Other relationally-complete formalisms

The online appendix [32] compares Relational Diagrams to other related visualizations like DFQL (Dataflow Query Language) [10, 14]. On a high level, all visual formalisms that we are aware of and that were proven to be relationally complete (including those listed in [10]) are at their core visualizations of relational algebra operators. This applies even to the more abstract *graph data structures (GDS)* from [9] and the later *graph model (GM)* from [11], which are related to our concept of *query representation.* The key difference is that GDS and GM are formulated inductively based on mappings onto operators of relational algebra. They thus mirror dataflow-type languages where visual symbols (directed hyperedges) represent operators like *set difference* connecting two relational symbols, leading to a new third symbol as output. Even QBE [57] uses the query pattern from RA and Datalog$^\neg$ of implementing relational division (or universal quantification) in a dataflow-type, sequential manner. Similarly, SIEUFERD [7], a direct manipulation spreadsheet-like interface, uses direct translation of relational algebra operators to prove SQL-92 completeness. This

[9]Every beta graph has lines, and graphs with lines but no predicates have meanings. See, e.g., the definition in [50, p. 41].

translation involves expressing set difference with outer joins and "IS NULL" conditions. We have proved that there are simple queries in relational calculus that cannot be represented in relational algebra with the same number of relational symbols. Thus any visual formalism based on relational algebra cannot represent the full range of relational query patterns.

7.4 Other diagrammatic and non-diagrammatic query representations

Visual SQL [38] is a visual query language that also supports query visualization. With its focus on query specification, it maintains the one-to-one correspondence to SQL, and syntactic variants of the same query lead to different representations. SQLVis [41] shares motivation with QueryVis. Similar to Visual SQL, it places a stronger focus on the actual syntax of a SQL query and syntactic variants like nested EXISTS queries change the visualization, and join conditions are expressed as text. StreamTrace [8] focuses on visualizing temporal queries with workflow diagrams and a timeline. It is an example of visualizations for spatiotemporal domains and not the logic behind general relational queries. DataPlay [2, 3] allows a user to specify their query by interactively modifying a query tree with quantifiers and observing changes in the matching/non-matching data. It does not have a union operator and is thus not relationally complete. For a more detailed discussion we refer to two recent tutorials on visual representations of relational queries [30, 31].

8 CONCLUSIONS AND FUTURE WORK

We motivated a criterion called *pattern-isomorphism* that captures the patterns across relational languages and gave evidence for its importance in designing diagrammatic representations. We formulated the non-disjunctive fragments of Datalog$^{\neg}$, RA, safe TRC, and corresponding SQL (interpreted under set semantics) that naturally generalize conjunctive queries to nested queries with negation. We prove that this important fragment allows a rather intuitive and, in hindsight, natural diagrammatic representation that can preserve the query pattern used across all four languages. We further prove that this formalism, extended with a representation of union, is complete for full safe relational calculus (though not pattern-preserving) and showed via user studies strong evidence that this diagrammatic representation allows users to understand query patterns faster and more accurately than SQL, even with minimal training.

Finding a *pattern-preserving* diagrammatic representation for disjunction and even more general features of SQL (such as grouping and aggregates) is an open problem. For example, it is not clear how to achieve an intuitive and principled *diagrammatic* representation for arbitrary nestings of disjunctions, such as "$R.A < S.E \wedge (R.B < S.F \vee R.C < S.G)$" or "$(R.A > 0 \wedge R.A < 10) \vee (R.A > 20 \wedge R.A < 30)$" with minimal additional notations. Grounded in a long history of diagrammatic representations of logic, we gave intuitive arguments for why visualizing disjunctions is inherently more difficult than conjunctions, with some experts believing it is not possible [49, 50] unless one adds *non-diagrammatic abstractions*.

ACKNOWLEDGMENTS

This work was supported in part by a Khoury seed grant program, 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. We like to thank Mirek Riedewald for helpful comments on an early version of this paper, and Jan Van den Bussche for very insightful comments about an earlier version of the proof of separation lemma 20 in the online appendix [32].

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley. http://webdam.inria.fr/Alice/

[2] Azza Abouzied, Joseph M. Hellerstein, and Avi Silberschatz. 2012. DataPlay: interactive tweaking and example-driven correction of graphical database queries. In *UIST.* ACM, 207–218. https://doi.org/10.1145/2380116.2380144

[3] Azza Abouzied, Joseph M. Hellerstein, and Avi Silberschatz. 2012. Playful Query Specification with DataPlay. *PVLDB* 5, 12 (2012), 1938–1941. https://doi.org/10.14778/2367502.2367542

[4] Micah Allen, Davide Poggiali, Kirstie Whitaker, Tom Rhys Marshall, Jordy van Langen, and Rogier A. Kievit. 2019. Raincloud Plots: a multi-platform tool for robust data visualization. *Wellcome Open Research 4* (2019). https://doi.org/10.12688/wellcomeopenres.15191.2

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

[6] Marcelo Arenas, Pablo Barcelo, Leonid Libkin, Wim Martens, and Andreas Pieris. 2022. *Database Theory: Querying Data.* Open source at https://github.com/pdm-book/community.

[7] Eirik Bakke and David R. Karger. 2016. Expressive Query Construction through Direct Manipulation of Nested Relational Results. In *SIGMOD.* ACM, 1377–1392. https://doi.org/10.1145/2882903.2915210

[8] Leilani Battle, Danyel Fisher, Robert DeLine, Mike Barnett, Badrish Chandramouli, and Jonathan Goldstein. 2016. Making Sense of Temporal Queries with Interactive Visualization. In *CHI.* ACM, 5433–5443. https://doi.org/10.1145/2858036.2858408

[9] Tiziana Catarci. 1991. On the Expressive Power of Graphical Query Languages. In *Visual Database Systems, II. Proceedings of the IFIP TC2/WG 2.6 Second Working Conference on Visual Database Systems. (IFIP Transactions, Vol. A-7).* North-Holland, 411–421. https://dblp.org/rec/conf/vdb/Catarci91

[10] Tiziana Catarci, Maria Francesca Costabile, Stefano Levialdi, and Carlo Batini. 1997. Visual Query Systems for Databases: A Survey. *J. Vis. Lang. Comput.* 8, 2 (1997), 215–260. https://doi.org/10.1006/jvlc.1997.0037

[11] Tiziana Catarci, Giuseppe Santucci, and Michele Angelaccio. 1993. Fundamental Graphical Primitives for Visual Query Languages. *Inf. Syst.* 18, 2 (1993), 75–98. https://doi.org/10.1016/0306-4379(93)90006-M

[12] Stefano Ceri and Georg Gottlob. 1985. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Trans. Software Eng.* 11, 4 (1985), 324–345. https://doi.org/10.1109/TSE.1985.232223

[13] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *PVLDB* 11, 11 (2018), 1482–1495. https://doi.org/10.14778/3236187.3236200

[14] Gard J. Clark and C. Thomas Wu. 1994. DFQL: Dataflow query language for relational databases. *Inf. Manag.* 27, 1 (1994), 1–15. https://doi.org/10.1016/0378-7206(94)90098-1

[15] Thomas M. Connolly and Carolyn E. Begg. 2015. *Database Systems: A Practical Approach to Design, Implementation and Management, Global Edition* (5 ed.). Pearson Addison Wesley. https://www.pearson.com/en-gb/subject-catalog/p/database-systems-a-practical-approach-to-design-implementation-and-management-global-edition/P200000003964/

[16] Michael Correll. 2023. Teru Teru Bozu: Defensive Raincloud Plots. *Computer Graphics Forum (EuroVis)* 42, 3 (2023), 235–246. https://doi.org/10.1111/cgf.14826

[17] Geoff Cumming. 2013. *Understanding the new statistics: Effect sizes, confidence intervals, and meta-analysis.* Routledge. https://doi.org/10.4324/9780203807002

[18] Jonathan Danaparamita and Wolfgang Gatterbauer. 2011. QueryViz: Helping Users Understand SQL queries and their patterns. In *EDBT.* ACM, 558–561. https://doi.org/10.1145/1951365.1951440

[19] Christopher J. Date. 2003. *An introduction to database systems* (8 ed.). Pearson/Addison Wesley Longman. https://dl.acm.org/doi/10.5555/861613

[20] Jan Van den Bussche and Stijn Vansummeren. 2009. Translating SQL into the relational algebra. Course notes, Hasselt University and Université Libre de Bruxelles. https://dipot.ulb.ac.be/dspace/bitstream/2013/198813/1/sql2alg_eng.pdf

[21] Sara Di Bartolomeo, Mirek Riedewald, Wolfgang Gatterbauer, and Cody Dunne. 2021. STRATISFIMAL LAYOUT: A modular optimization model for laying out layered node-link network visualizations. *IEEE Transactions on Visualization and Computer Graphics (VIS'21)* 28, 1 (2021), 324–334. https://doi.org/10.1109/TVCG.2021.3114756 Preprint & Supplemental Material: https://osf.io/qdyt9.

[22] Pierre Dragicevic. 2016. *Fair Statistical Communication in HCI.* Springer International Publishing, Cham, 291–330. https://doi.org/10.1007/978-3-319-26633-6_13

[23] Pierre Dragicevic. 2018. Can we call mean differences "effect sizes"? https://transparentstatistics.org/2018/07/05/meanings-effect-size/

[24] Bradley Efron. 1987. Better Bootstrap Confidence Intervals. *J. Amer. Statist. Assoc.* 82, 397 (1987), 171–185. https://doi.org/10.1080/01621459.1987.10478410

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

[26] Cibele Freire, Wolfgang Gatterbauer, Neil Immerman, and Alexandra Meliou. 2015. The Complexity of Resilience and Responsibility for Self-Join-Free Conjunctive Queries. *PVLDB* 9, 3 (2015), 180–191. https://doi.org/10.14778/2850583.2850592

[27] Jean H Gallier. 2011. *Discrete mathematics*. Springer. https://doi.org/10.1007/978-1-4419-8047-2

[28] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database systems: The complete book* (2 ed.). Prentice Hall Press. https://dl.acm.org/doi/book/10.5555/1450931

[29] Wolfgang Gatterbauer. 2011. Databases will Visualize Queries too. *PVLDB* 4, 12 (2011), 1498–1501. https://doi.org/10.14778/3402755.3402805

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

[31] Wolfgang Gatterbauer. 2024. A Comprehensive Tutorial on over 100 Years of Diagrammatic Representations of Logical Statements and Relational Queries. In *ICDE*. IEEE. Tutorial page: https://northeastern-datalab.github.io/diagrammatic-representation-tutorial/.

[32] Wolfgang Gatterbauer and Cody Dunne. 2023. Supplemental material for "On the reasonable effectiveness of Relational Diagrams". Homepage: https://relationaldiagrams.com/. Main suplemental 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/.

[33] 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

[34] Wolfgang Gatterbauer and Dan Suciu. 2014. Oblivious bounds on the probability of Boolean functions. *TODS* 39, 1 (2014), 5:1–5:34. https://doi.org/10.1145/2532641

[35] Paruntungan Girsang. 1994. *The comparison of SQL, QBE, and DFQL as query languages for relational databases*. Master's thesis. Naval Postgraduate School, Monterey, California. https://core.ac.uk/download/pdf/36723678.pdf

[36] Alon Y. Halevy, Peter Norvig, and Fernando Pereira. 2009. The Unreasonable Effectiveness of Data. *IEEE Intell. Syst.* 24, 2 (2009), 8–12. https://doi.org/10.1109/MIS.2009.36

[37] Joseph Y. Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. 2001. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic* 7, 2 (2001), 213–236. https://doi.org/10.2307/2687775

[38] Hannu Jaakkola and Bernhard Thalheim. 2003. Visual SQL – High-Quality ER-Based Query Treatment. In *ER (Workshops) (LNCS)*. Springer, 129–139. https://doi.org/10.1007/978-3-540-39597-3_13

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

[40] Wo-Shun Luk and Steve Kloster. 1986. ELFS: English Language from SQL. *ACM Trans. Database Syst.* 11, 4 (dec 1986), 447–472. https://doi.org/10.1145/7239.384276

[41] Daphne Miedema and George Fletcher. 2021. SQLVis: Visual Query Representations for Supporting SQL Learners. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–9. https://doi.org/10.1109/VL/HCC51201.2021.9576431

[42] Richard E. Pattis. 2013. EBNF: A Notation to Describe Syntax. https://ics.uci.edu/~pattis/misc/ebnf2.pdf. (accessed on September 21, 2021).

[43] Charles Sanders Peirce. 1933. *Collected Papers*. Vol. 4. Harvard University Press. https://doi.org/10.1177/000271623417400185

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

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

[46] Phyllis Reisner, Raymond F. Boyce, and Donald D. Chamberlin. 1975. Human Factors Evaluation of Two Data Base Query Languages: Square and Sequel. In *AFIPS (AFIPS '75)*. ACM, 447–452. https://doi.org/10.1145/1499949.1500036

[47] Relational Diagrams. 2023. https://www.relationaldiagrams.com.

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

[49] Sun-Joo Shin. 1995. *The Logical Status of Diagrams*. Cambridge University Press. https://doi.org/10.1017/CBO9780511574696

[50] Sun-Joo Shin. 2002. *The Iconic Logic of Peirce's Graphs*. The MIT Press. https://doi.org/10.7551/mitpress/3633.001.0001

[51] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts* (7 ed.). McGraw-Hill Book Company. https://www.db-book.com/db7/index.html

[52] Soufflé. 2023. https://souffle-lang.github.io/rules.

[53] Etienne Toussaint, Paolo Guagliardo, Leonid Libkin, and Juan Sequeda. 2022. Troubles with Nulls, Views from the Users. *PVLDB* 15, 11 (2022), 2613–2625. https://www.vldb.org/pvldb/vol15/p2613-guagliardo.pdf

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

[55] Eugene Wigner. 1960. The Unreasonable Effectiveness of Mathematics in the Natural Sciences. *Communications in Pure and Applied Mathematics* 13, 1 (1960), 1–14. https://doi.org/10.1002/cpa.3160130102

[56] Leland Wilkinson. 1999. Dot Plots. *The American Statistician* 53, 3 (1999), 276–281. https://doi.org/10.1080/00031305.1999.10474474

[57] Moshé M. Zloof. 1977. Query-by-Example: A Data Base Language. *IBM Systems Journal* 16, 4 (1977), 324–343. https://doi.org/10.1147/sj.164.0324

Received July 2023; revised October 2023; accepted November 2023