Principles of Query Visualization

Wolfgang Gatterbauer
Northeastern University
w.gatterbauer@northeastern.edu

Cody Dunne
Northeastern University
c.dunne@northeastern.edu

H.V. Jagadish
University of Michigan
jag@umich.edu

Mirek Riedewald
Northeastern University
m.riedewald@northeastern.edu

Abstract

Query Visualization (QV) is the problem of transforming a given query into a graphical representation that helps humans understand its meaning. This task is notably different from designing a Visual Query Language (VQL) that helps a user compose a query. This article discusses the principles of relational query visualization and its potential for simplifying user interactions with relational data.

1 What is Query Visualization (QV) and what is it for?

The design of relational query languages and the difficulty for users to compose relational queries have received much attention over the last 40 years [12,14,36,42,51,60,79,80,94,97]. A complementary and much-less-studied problem is that of helping users *read and understand an existing query*. Reading code is hard, and SQL is no exception. With the proliferation of public data sources, and associated queries, users increasingly have a need to read other people's queries and scripts. Furthermore, it is usually much easier to modify a draft than to write something from scratch. As such, modifying an already existing query could be an effective way to write new queries. However, modifying an existing query requires first to understand it (Figure 2). For these reasons, it is valuable to help users understand queries, and visualization is one obvious route. In this paper, we study the problem of query visualization with a view towards improving query understanding.

Consider the following five scenarios that illustrate how query visualizations can assist users:

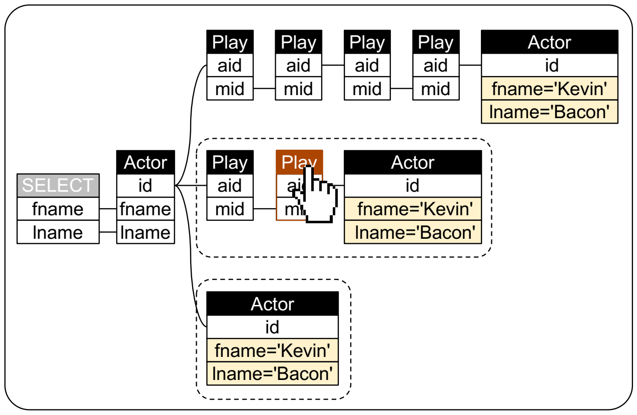
Scenario 1 (Data scientists reusing past queries): A group of data analysts are collaboratively analyzing movie data. This data is stored in a shared data repository. In addition to the data itself, they are also sharing their queries using a Collaborative Query Management System (CQMS) [4,5,16,24,26,44,49,54,55,62,65,70]. The query recommendation component of that tool suggests relevant previously-issued queries that the user can choose from, rather than write a query from scratch. The tool shows the queries both in text (Figure 1a) and as query visualization (Figure 1b). The visualization preserves the logical structure of the textual query. There is also a one-to-one mapping between the query and its visualization: As the user moves the mouse over components of the visualization, the corresponding component in the textual query is highlighted (and v.v.). The particular query used in Figure 1 is over the IMDB movie database and finds all actors with Kevin Bacon number 2 (i.e. actors who have not played in a movie with Kevin Bacon directly, but who have played with other actors who have played with Kevin Bacon).[1] These diagrams require some training to understand (see our discussion in section 3

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1]See https://youtu.be/kVFnQRGAQls?t=170 for an animated explanation of how to read and understand this particular query.

```
select distinct A0.fname, A0.lname
from Actor A0, play P0, Play P1, Play P2,
     Play P3, Actor A3
where A3.fname='Kevin' and A3.lname='Bacon'
and P3.aid=A3.id and P3.mid=P2.mid
and P2.aid=P1.aid and P1.mid=P0.mid
and P0.aid=A0.id
and not exists (select *
  from Actor A4, Play P4, Play P5
  where A4.fname='Kevin' and A4.lname='Bacon'
  and A4.id=P4.aid and P4.mid=P5.mid
  and P5.aid=A0.id)
and not exists (select *
  from Actor A5
  where A5.fname='Kevin' and A5.lname='Bacon'
  and A5.id=A0.id)
```

(a) SQL query



(b) Query Visualization

Figure 1: Scenario 1: A user searches for a query by browsing through a repository of previously-recorded SQL queries. For each query, she needs to *quickly understand* its meaning. A query visualization panel helps her understand the query by showing a succinct representation of its relational query pattern. The shown query returns actors with Bacon number 2. As she hovers her mouse over parts of the query, both the textual and visualized query highlight corresponding parts in synchronization.

for how to read them). However, in a controlled and pre-registered user study (see Section 4.1) we found that even a few minutes of training suffice, and experienced SQL users could interpret queries *in less time* and *with fewer errors* using these diagrams instead of using SQL alone [61].

A widely known example of such a shared data and query repository is the Sloan Digital Sky Survey (SDSS) [4, 87]. Data about stars was put into a relational database and is freely available for access. Astronomers, who are mostly “hobbyist" SQL users, have to write queries to get the data they want. In most cases, the queries they wish to write are similar to queries others have written before. So the standard workflow is for the user to look at previously issued queries, find one that is close to what they want, and then modify it to suit their analysis needs. In response, SDSS has added templates for commonly written query types.[2]

Scenario 2 (Visual feedback during query editing): As the user starts editing the SQL query (Figure 1a), the query visualization gets updated too (Figure 1b, updates are not shown). When a syntactically-correct query does not give the expected result, the query visualization can help the user understand that an incorrect join pattern was used between the various subqueries.

Scenario 3 (Learning from galleries of relational query patterns): A data scientist wants to issue another query and looks for inspiration in a *web gallery of SQL design patterns*. Similar to the way users of Matplotlib[3], D3[4], and Altair[5] program new visualizations by browsing through, copying from, and adapting existing designs [7], such galleries enhance the technical skills of data scientists and learners by showing a range of possible relational patterns and design templates to learn from that would be hard to browse and make sense of based on text alone. Similar programmer behaviors are found outside of visualization, where existing code templates, examples, and idioms are extensively copied and adapted. From IDE (Integrated Development Environment) logs of 81 developers, Ciborowska et al. [19] identified many cases of opportunistic code reuse from the Web followed by

[2]http://skyserver.sdss.org/dr8/en/help/docs/realquery.asp
[3]https://matplotlib.org/stable/gallery/index.html
[4]https://d3-graph-gallery.com/
[5]https://altair-viz.github.io/gallery/index.html

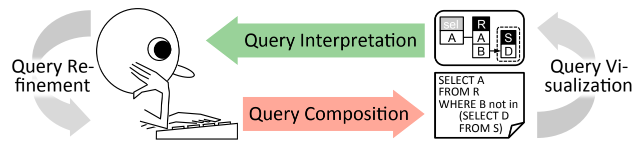


Figure 2: The goal of query visualization is to complement (but not substitute) the composition of queries by creating automatic visualizations of queries. Composition of a query is still performed via unambiguous and expressive text. The transformation from text into a visualization can abstract away from a concrete syntax and thus be non-injective. Compare to the write-format-preview cycle used by LaTeX [57] in that a user writes text, the system then autoformats and renders a document, which the user can then peview.

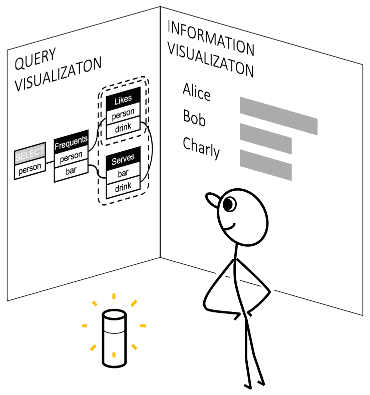


Figure 3: Scenario 5: An analyst dictates queries to her voice assistant which then shows the query as understood together with the query answers.

editing the code. LaToza et al. [58] surveyed 157 programmers, and 56% agreed that understanding code that someone else wrote is a serious problem. Yang et al. [96] found that many blocks of Python code are copied from Stack Overflow into open-source projects with slight modifications. Ahmed et al. [3] found that 24% of copy-and-paste events among 21,770 users of Eclipse were from sources external to the IDE, though this is likely an overestimate. Brandt et al. [10] found that such copy-and-paste programming is particularly beneficial for programmers working in new domains: In their study of students learning to use a new framework, one-third of the participants' code consisted of modified versions of examples from the documentation.

Scenario 4 (Clustering pattern-identical queries): A teacher receives the SQL solutions for a homework from her 50 students. An automatic correction tool, such as ADUSA [52], Cosette [18], Qex [92] TATest [67], or XData [15], determines that 40 of those solutions are correct. But those correct solutions "look" very different, even after applying some standard SQL pretty printer, such as sqlparse [90]. The queries use different table aliases, nesting patterns, join sequences, and at times different syntactic constructs, such as implicit joins in the WHERE clause or infixed SQL-92 join notation. The teacher would like to cluster the 40 correct solutions not by their syntactic variants, but by whether there are 'truly' *novel patterns* beyond the 3 she currently knows. Query visualization makes it easier to cluster and compare the 40 queries, because a good visualization captures the essence of the query structure, abstracting away superficial syntactic differences. And, indeed, the teacher finds 2 new patterns. She can now show and discuss 5 total patterns with the learners in the next class.

Scenario 5 (Visual feedback from voice assistants): We now switch to the year 2045. A data analyst stands in her office analyzing some company data. She directs possible queries to her voice assistant which then visualizes on the walls the queries together with the data (Figure 3). The visualization of the query provides immediate feedback on what the assistant understood.

Common to these 5 scenarios is that query visualization helps the users achieve new functionalities or increased efficiency in composing queries as a *complement* to query composition and *not a substitute* for it.

Definition 1 (Query Visualization): The term "query visualization" refers to both (i) a graphical representation of a query and (ii) the process of transforming a given query into a graphical representation. The goal of query visualization is to help users more quickly understand the intent of a query, as well as its relational query pattern.

When we say "query visualization", we will typically mean the end result. From context, it should be clear to the reader on the few occasions when we mean the process rather than the end result.

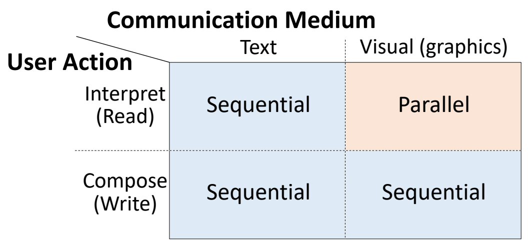


Figure 4: *Composing* a query with a visual query language is as sequential as composing it with SQL. *Interpreting* a visualization (whether of information or a query) is the only modus in which a user can act on information in parallel, leveraging the speed of the human perceptual system (orange = easier, blue = harder).

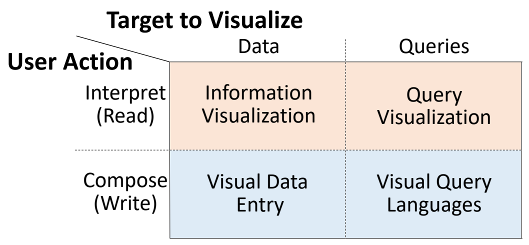


Figure 5: *Visual Query Languages* allow a user to compose queries. They have been widely studied and have a rich history. In contrast, *Query Visualization* helps the user understand an existing query just as *Information Visualization* helps understand data (orange = easier, blue = harder).

2 What Query Visualization is not

2.1 Query Visualization is not the same as a Visual Query Language (VQL)

Visual Query Languages (VQLs) provide languages to express queries in a visual format. Visual Query Systems (VQSs) implement VQLs and generate queries from visual representations constructed by users [11]. Such visual methods for specifying relational queries have been studied extensively (a 1997 survey by Catarci et al. [12] cites over 150 references), and many commercial database products offer some visual interface for users to write SQL queries. In parallel, there is a centuries-old history on the study of formal diagrammatic reasoning systems [45] with the goal of helping humans to reason in terms of logical statements.[6] Yet despite their extensive study and intuitive appeal, successful visual tools today mostly only *complement instead of replace* text for specifying queries. Why has visual query specification not yet replaced textual query specification?

We believe that there are two primary reasons: (1) First, humans are *better in interpreting rather than composing visuals* because visual composition is an inherently sequential process (Figure 4). All human input methods (composition) are sequential, whether resulting in text or a graphic. Visual perception is a remarkable human sense (interpretation) that can understand inputs in parallel, and it works dominantly by *spatial arrangement of information*. While reading text is also a visual activity, the spatial arrangement of the letters requires a sequential scan of the text (though notice that pretty printers can spatially arrange text, Section 2.4). Hence, visual interpretation of graphics is the fastest way to communicate with humans, and it only works well for understanding rather than composing. Even in theory, there is no dramatic speed-up in using a visual language for composition. In practice, the user interaction is quite cumbersome: users must be able to interactively construct and manipulate expressions in a visual language and connect graphical elements to establish graphical relationships. In turn, the program must provide appropriate interpretations of mouse, touch, and keyboard events, and it is difficult to build formal grammars and compilers for two-dimensional drawing areas. In sum, solutions to these graphical requirements are intricate, inherently difficult to implement, and challenging to use [98].

(2) A second reason is that graphs are more ambiguous than text, i.e. it is more difficult to be precise with a visual representation than with text. In order to precisely specify a query, possible options and specific details affecting query semantics must be presented. In contrast, understanding a query requires a focus on the high-level structure, abstracting away low-level details and subtleties. In programming languages, this distinction is clearly made between *visual programming* for developing a program and *program visualization* for analyzing an existing program [72].

[6]A relational query is a logical formula with free variables. A logical statement has no free variables and is intuitively the same as a Boolean query that returns a truth value of TRUE or FALSE.

This leads us to suggest a user-query interaction that separates the query composition from the visualization (Figure 2): Composition is unchanged and best done in text (or alternatively with exploratory input formats like natural language). But composition is augmented and *complemented with a visual that helps interpretation*. Recall Scenario 5 where a digital voice assistant connects to omnipresent screens to show what it understood before executing a command (Figure 3). Compare to the way that many of us write research papers: we write LaTeX code in a text editor but prefer to read and verify the auto-compiled PDF using automatic or editor-specific build instructions (Figure 2).

Finally notice that query visualization is related to *Information Visualization* [17], which also focuses on helping users understand complex relationships, but in data instead of in query logic (Figure 5).

2.2 Query Visualization is not the same as Query Plan Visualization

Readers may be familiar with visualizations of query plans. Figure 6a shows a query plan chosen by PostgreSQL [77] to run query Q_{some} from Figure 7a (*Find persons who frequent some bar that serves a drink they like*). Similarly, Figure 6b shows the same query expressed in DFQL (Dataflow Query Language) [20] which is modeled after relational algebra. Notice that neither visualization captures the cyclic nature of the joins in query Q_{some}. A query plan visualization attempts to represent HOW a query is executed. In contrast, a query visualization attempts to represent WHAT a query does (i.e. its intent) and possibly the relational pattern it uses. See the query visualization in Figure 7b which shows the join pattern and that this query is cyclic. Similarly, query visualizations are also different from visualizing and comparing the cost or speed of execution plans [43].

2.3 Query Visualization is only partially related to Visal Query Debugging

An important reason for why we want to help users understand HOW exactly a given query is executed is for debugging a faulty query. Visualizing software execution behavior can be helpful for program debugging [30, 81], but only if it helps explain WHY a query returns a particular result or WHY NOT [66]. To achieve this more fine-grained understanding, state-of-the-art workflows for debugging SQL queries help users understand queries by somehow *showing intermediate results* [37, 38, 67, 74]. Thus it is helpful to see debugging as a spectrum of goals with different "granularities." At one end of the spectrum, a query may be faulty because two incorrect tables are joined (a tool that answers WHAT the query actually does would help here). At the other more-fine grained level, a query may be faulty because a DMBS implemented a particular SQL syntax for handling NULLS incorrectly[7], and it seems there is no way to avoid using data examples for effective debugging.

2.4 Alternatives to Query Visualization for helping users understand existing queries

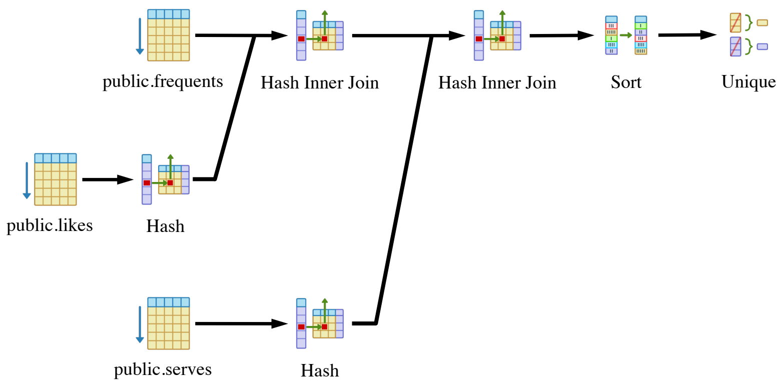
There are three main alternative approaches for helping users understand existing queries:

(1) Illustrating queries by examples. Several papers suggest illustrating the semantics of operators in a data flow program or the semantics of queries by generating *example input and output data*, and possibly *intermediate data*. The result is basically a list of tuples for each relational operator [1, 15, 37, 38, 53, 67, 74].

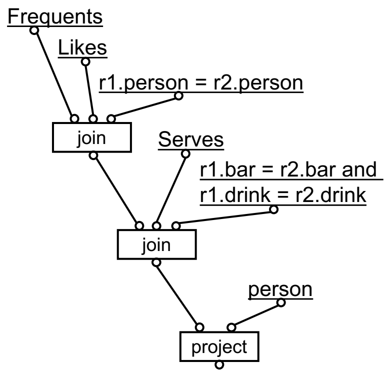
(2) Translating queries into Natural Language (NL). Translating between SQL and NL is a heavily researched topic, and various ideas are proposed to explain queries in NL [34, 47, 56, 86, 95]. Work in this area convincingly argues that automatically creating effective free-flowing text from queries is difficult and that the overall task is quite different from previous work on creating NL interfaces to DBMSs [50]. There is also recent work on translating query plans into NL [93].

(3) Pretty printing queries. Query editors for major DBMSs use *syntax highlighting and aligning* of query blocks and clauses. Pretty printers, such as sqlparse [90], automatically arrange a SQL query in a supposedly easy-to-read form. The most important dimensions are colors, capital vs. small letters, and indentation.

[7]As example see https://stackoverflow.com/questions/19686262/query-featuring-outer-joins-behaves-differently-in-oracle-12c



(a) Postgres EXPLAIN command for Q_{some}



(b) Q_{some} in DFQL

Figure 6: Query visualizations are not query plans nor data flow diagrams: (a) Visualized query plan by Postgres' EXPLAIN command [77] for query Q_{some} from Figure 7a. (b) Same query expressed in DFQL (Dataflow Query Language) [20] which is modeled after relational algebra. Notice that neither visualization captures the cyclic nature of the joins (see Figure 7b). "*Query visualization*" to "*query plan visualization*" is the same as "*intend of a query* (WHAT)" to "*execution of a query* (HOW)".

A key difference of alternatives to query visualization is that they are all inherently linear. A list of tuples or a textual description do not readily reveal common logical pattern behind queries. In particular, we are not aware of any SQL to NL tool available today that could translate our example from Figure 9 into an intuitive NL representation. Patterns are naturally best shown visually, and even the programming design patterns book [29] illustrates its patterns with intuitive diagrams. A theory of relational query patterns, and a query-user interaction pattern inspired by "mix-and-match", seems naturally supported by a visual approach. In addition, our recent work [33] has shown that certain types of relational patterns cannot be represented in an operator-style (thus sequential) model.

3 Principles of Query Visualization and Design trade-offs

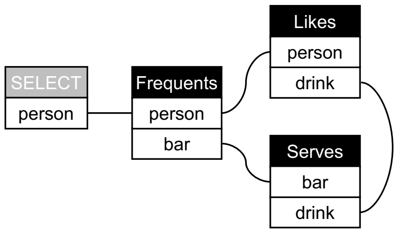
The challenge of query visualization is to find appropriate visual metaphors that (i) allow users to quickly understand a query's intent, even for complex queries, (ii) can be easily learned by users, and (iii) can be obtained from SQL by automatic translation, including a visually-appealing automatic arrangement of nodes of the visualization. We believe that—with the right visual alphabet—users can learn to interpret visualized queries by seeing examples without much active focus. This is similar to what is known in language learning theory as the difference between the active and the generally larger passive vocabulary: Actively reproducing newly learned content is generally more difficult than passively recognizing such content.

We next discuss the principles that led us to a particular design of a query visualization language (actually two variants, which we discuss later in more detail). We list those here to spark a healthy debate. Not all listed principles are universal, and deviations may lead to interesting alternative design decisions. These principles are also not MECE (Mutually Exclusive and Collectively Exhaustive), and some design decisions can be justified separately from other overlapping decisions.

(1) Existing metaphors as starting point: Ideally, a query visualization can be learned "on-the-fly" by seeing visualizations of increasing complexity, starting from examples that are already familiar. Most database users are familiar with the UML diagram notation for classes and their attributes [28] applied to database schemas: Table names on top of column names in rectangular bounding boxes, primary-foreign keys contraints represented by lines between column names. The visualization of a conjunctive query should thus not depart too much from

```
select distinct F.person
from Frequents F, Likes L, Serves S
where F.person = L.person
and F.bar = S.bar
and L.drink = S.drink
```

(a) Q_{some} in SQL

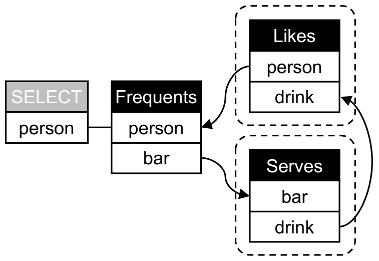


(b) Q_{some} in QueryVis

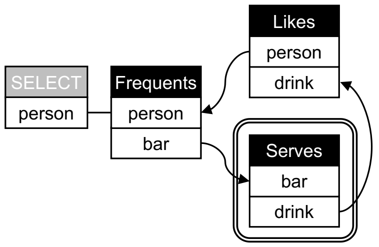
Figure 7: Principles 1 & 2: Visualizing a conjunctive query should follow a familiar UML notation: *Find persons who frequent some bar that serves some drink they like*. The only novelty is a dedicated output table on the left, emphasizing the compositionality of the relational model, and supporting an output-oriented reading order.

```
select distinct F.person
from Frequents F
where not exists
    (select *
    from Serves S
    where S.bar = F.bar
    and not exists
        (select L.drink
        from Likes L
        where L.person = F.person
        and S.drink = L.drink))
```

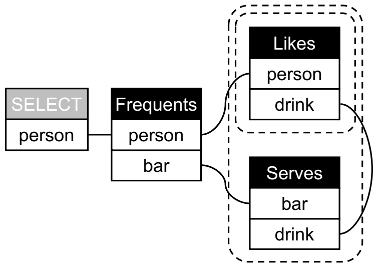
(a) Q_{only}



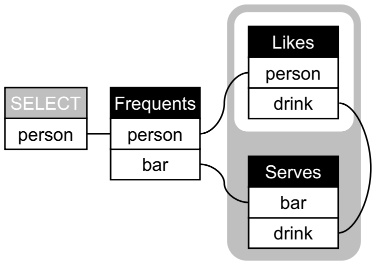
(b) Q_{only}



(c) Q_{only}



(d) Q_{only}



(e) Q_{only}

Figure 8: Principles 3 & 8: (a) *Find persons who frequent some bar that serves ONLY drinks they like*. QueryVis: (b) Visualizing a nested query still follows familiar UML notations, but now adds visual metaphors for $\nexists$ (dashed box) and the reading order can be found by following the arrows. (c) The reading can be further simplified by the use of the $\forall$ quantifier (double-lined bounding box), a logical and intuitive operator that does not exist in SQL. The visualization asks for *persons who frequent some bar so that ALL drinks served are liked by them*. Relational Diagrams are an alternative visualization that replaces arrows and reading orders by explicit enclosure to express nesting relationships (d) and (e).

such deeply familiar visual metaphors (e.g., see the conjunctive query Q_{some} in Figure 7a and its visualization in Figure 7b). More complicated queries then progressively extend such familiar visual metaphors.

(2) Compositionality of the relational model: Inputs to queries are tables, and the output of a query is another table. Visualizations can (and we think should) emphasize this compositionality by explicitly showing an output table. This compositionality is also illustrated by the Relational Tuple Calculus expression for Figure 7a:

$$\{q(\mathit{person}) \mid \exists f \in \mathit{Frequents}, \exists l \in \mathit{Likes}, \exists s \in \mathit{Serves}[q.\mathit{person} = f.\mathit{person} \wedge \\ f.\mathit{person} = l.\mathit{person} \wedge l.\mathit{drink} = s.\mathit{drink} \wedge s.\mathit{bar} = f.\mathit{bar}]\}$$

The expression makes use of 4 tables: 3 input tables (*Frequents*, *Likes*, and *Serves*), and 1 output table called q (Figure 7b names the output table "SELECT"). In contrast, all interactive query tools listed in section 5 use either checkmarks, stars, or colors to highlight a subset of attributes that are returned by the query.

(3) Progressive visual complexity: Entropy codes, such as Huffman codes [21], compress data by encoding symbols with an amount of bits inversely proportional to the frequency of the symbols. In the same spirit, a visual alphabet should be adapted to an overall expected workload and visual constructs for more common logical operators should be designed with lower visual complexity than less common ones. Starting from UML and its familiar notations for schemas and conjunctive queries, we can then enhance the visual representation in a *progressive way*. For example, almost all database queries use the logical AND in their first-order logic translation (e.g. joins, EXISTS, IN), but only few use OR (e.g. OR, UNION). If infrequent query constructs become increasingly complex to read, this progression does not decrease the overall usability, but rather assures that more often used constructs are simple to read, in turn. For example, the visualization of the query from Figure 8a is expected to be at least as "complicated" as the query from Figure 7a.

For increasing complexity of nested queries with negation, we are inspired by a body of work on *diagrammatic reasoning systems* [45]. Diagrammatic notations are in turn inspired by the influential *existential graph notation* by Charles Sanders Peirce [75, 82, 85]. These graphs exploit topological properties, such as enclosure, to represent logical expressions and set-theoretic relationships (see description in Figure 8).

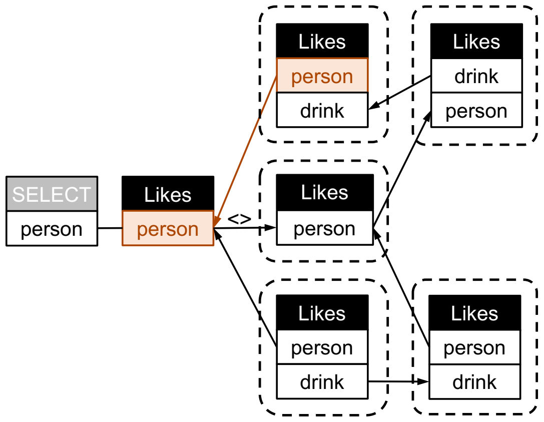
(4) Expose (and not hide) relational patterns: We believe that a query visualization should expose the relational pattern used in a textual query, instead of replacing it with an abstraction and concepts that go beyond the relational model. This requires a visualization to use the same number of input tables of the textual query and to preserve a 1-to-1 mapping between them. To illustrate, consider the SQL query in Figure 9a asking for "persons with a unique drink taste." The query uses 6 instances of the same table in a pattern that reads "return any person, s.t. there does not exist any other person, s.t. there does not exist any drink liked by that other person that is not also liked by the returned person and there does not exist any drink liked by the returned person that is not also liked by the same other person." The visualization in Figure 9b does not replace that relational pattern with another shorter construct, but rather makes it easier to inspect and reason about: it complements the textual query and preserves some traceable mapping between query and visualization. It preserves its relational pattern.

(5) Minimal visual complexity: A query visualization should fulfill some kind of minimality criteria. Intuitively, we aim to minimize the ink-data ratio (thus we like to maximize its inverse: Edward Tufte's famous data-ink ratio defined as the proportion of a graphic's "ink" devoted to the informative and thus non-redundant display of data information [91]). Minimality can be interpreted in different ways: For example, the visual alphabet could contain only a minimal set of different visual elements; removing an element would then render the visualization less expressive. Or, for a given query, removing a particular visual element would render the query incomplete. To achieve such minimality, one can take inspiration by comparative analysis of existing textual languages. For example, (i) Datalog does not require the use of table aliases (different occurrences of the same input relation can be distinguished by their join patterns), whereas SQL requires alias. On the other hand, (ii) SQL (inspired by tuple relational calculus) does not reference any attribute that is not used by the query, whereas Datalog uses positional information and thus requires to maintain positional information. Figure 9b shows an example QueryVis visualization that combines the best of both worlds: (i) repeated relations do not require aliases, and (ii) each of those occurrences only displays attributes needed.[8]

[8]A visualization could still show aliases to make it easier to maintain a static correspondence between the query and it visualization. However those aliases are not needed to interpret the meaning of a visual diagram.

```
select distinct L1.person
from Likes L1
where not exists
  (select *
  from Likes L2
  where L1.person <> L2.person
  and not exists
    (select  *
    from Likes L3
    where L3.person = L2.person
    and not exists
      (select *
      from Likes L4
      where L4.person = L1.person
      and L4.drink = L3.drink)
  and not exists
    (select *
    from Likes L5
    where L5.person = L1.person
    and not exists
      (select *
      from Likes L6
      where L6.person = L2.person
      and L6.drink = L5.drink)))
```

(a) SQL query



(b) Query Visualization

Figure 9: Principles 4 & 5: (a) Unique-set-query "*Find person with a unique drink taste*." (b) QueryVis diagram with reading order encoded by arrows (please see [61] for a detailed discussion of the query and its visualization). There is a 1-to-1 correspondence between the SQL query and its visualization. As the user moves the mouse over fragments of the query, the graphical representation highlights the corresponding visual elements.

(6) Abstract away from syntax details: A query visualization abstracts away from language-specific peculiarities. It can thus be non-injective with regard to syntactic redundancy. A prominent example is SQL's use of NULLs. While there has been a lot of work on putting SQL's use of NULL values on solid foundations, there is no universally agreed standard, and SQL queries evaluated on databases with NULL "may produce answers that are just plain wrong" [39]. The goal of query visualization can't be to provide an unambiguous interpretation of queries in the presence of NULLs, and thereby as a side-product also fix issues that have vexed database theoreticians over decades. Rather, the focus of query visualization on the underlying relational patterns means that query visualizations need to abstract away from such oddities and not preserve them (see Figure 10). Also, a query visualization is meant to complement a textual original query. It thus does not have to preserve all the information from the query; it can be non-injective, thereby dividing the work: a visualization for the overall pattern, the text for the details. This point goes back to section 2 and what query visualization does not try to achieve. The focus is on WHAT a query does, yet confined to the relational model and the particular underlying relational pattern (including all the input tables used), not the syntax nor the HOW of a particular execution plan. Tools that help users cope with the inherent syntactic difficulty of SQL fall under the category of SQL debugging (Section 2.3). The common foundation of all relational query languages and the relational model is First-Order Logic. We thus believe that focusing on the *logical interpretation* of queries [41] and set semantics provides a solid and well-understood foundation for query visualization. See Figure 10 for an example.

(7) Output-oriented reading order: Similar to a SQL query having an expected order of clauses (e.g. SELECT-FROM-WHERE), also a query visualization benefits from having an expected arrangement and reading order. Mirroring the design decision from SQL and calculus, we suggest a reading order left-to-right that starts with the result of the query (the output table) and then adds tables in horizontal layers in decreasing order of their relatedness to the output table (see Figure 10c). This suggests an arrangement where the input tables are placed at a horizontal distance from the output table that represents the shortest path join connection to the output table. Furthermore, the arrangement of the input tables should be such as to simplify the reading and understanding of the query by following aesthetic heuristics (e.g. minimizing the number of line crossings).

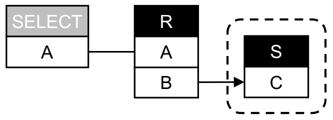
(8) Logic-based visual transformations: Nested negated quantifiers are particularly difficult to understand for users [79, 80]. Simple visualizations of logical transformation can further help show the query in a more

```
select distinct A
from R
where B not in
    (select C
    from S)
```

(a)

```
select distinct A
from R
where not exists
    (select *
    from S
    where B=C)
```

(b)



(c)

Figure 10: Principles 6 & 7: Queries (a) and (b) are equivalent except if column S.C contains NULL values. Thus ignoring NULL values in the database, they are equivalent and their *query intent* can be represented by the same QueryVis representation shown in (c) (example taken and slightly fixed from Fig. 4 in [31]).

intuitive form. Take a double negated query such as Figure 8:

$$\{q(person) \mid \exists f \in Frequents[q.person = f.person \wedge \neg\exists s \in Serves[s.bar = f.bar \wedge \neg\exists l \in Likes[l.drink = s.drink \wedge f.person = l.person]\}$$

The same query can be arguably understood more easily by writing it as

$$\{q(person) \mid \exists f \in Frequents[q.person = f.person \wedge \forall s \in Serves[s.bar = f.bar \rightarrow \exists l \in Likes[l.drink = s.drink \wedge f.person = l.person]\}$$

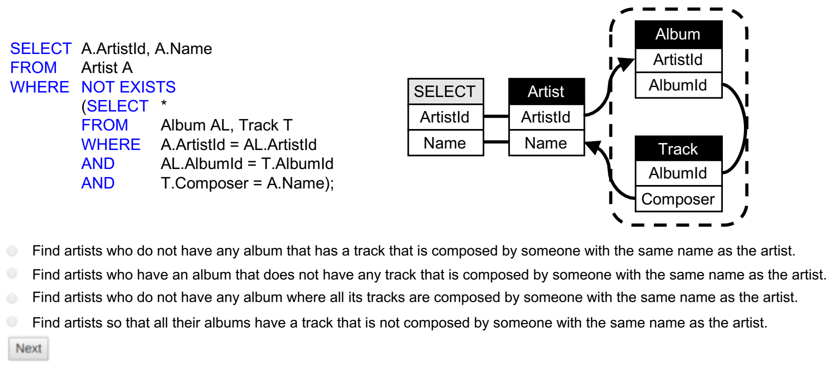
4 Our Suggestions: QueryVis and Relational Diagrams

When following the earlier listed design principles, a family of query visualizations naturally emerges. We discuss here two instances that differ in the way they visually encode the *nesting structure between query blocks*:

(1) QueryVis [22, 31, 61]: This earlier variant from 2011 borrows the idea of a "default reading order" from diagrammatic reasoning systems [27] and uses *arrows* to indicate an implicit reading order between different nesting levels. Take as example Figure 10c and notice how the arrows between the relations correspond to the order in which they appear in the natural language translation ("Find persons who FREQUENT some bar that SERVES only drinks they LIKE"). Without the arrows, there would be no natural order placed on the existential quantifiers and the visualization would be ambiguous. QueryVis focuses on the non-disjunctive fragment of relational calculus and is guaranteed to represent connected nested queries unambiguously up to nesting level 3. An interactive online version is available linked from https://queryvis.com.

(2) Relational Diagrams [33]: This more recent variant indicates the nesting structure of table variables by using *nested negated bounding boxes* instead of arrows. The nesting of negation boxes is more closely inspired by Peirce's influence beta existential graphs [75, 82, 85]. Interestingly, because Relational Diagrams are based on Tuple Relational Calculus (instead of Domain Relational Calculus which is closer to First-Order Logic) they solve interpretation problems of beta graphs that have been the focus of intense research in the diagrammatic reasoning communities. The big advantage of this variant is that it has a provably unambiguous interpretation for any nesting depth, even for queries with disconnected components, and for both Boolean and non-Boolean queries. Furthermore, by adding one additional visual element, Relational Diagrams can be made relationally complete even for non-Boolean queries. The downside is that these diagrams need more "ink" for simple nested queries, and logical transformations (design decision 8) cannot be as easily applied anymore. An alternative to such transformations, however, is using shading for alternating nesting depths (see e.g. Figure 8e).

An interactive online version of QueryVis has been online at https://queryVis.com since 2011 [22]. We encourage the reader to try it. It currently supports only a limited SQL grammar (see the web page for details). Still, this online demo shows that query visualization can have a very lightweight interaction. The user does not


```
SELECT  A.ArtistId, A.Name
FROM    Artist A
WHERE   NOT EXISTS
        (SELECT  *
        FROM     Album AL, Track T
        WHERE    A.ArtistId = AL.ArtistId
        AND      AL.AlbumId = T.AlbumId
        AND      T.Composer = A.Name);
```

- Find artists who do not have any album that has a track that is composed by someone with the same name as the artist.
- Find artists who have an album that does not have any track that is composed by someone with the same name as the artist.
- Find artists who do not have any album where all its tracks are composed by someone with the same name as the artist.
- Find artists so that all their albums have a track that is not composed by someone with the same name as the artist.

Figure 11: Section 4.1: Example query from our user study. The query is shown in the *Both* condition, in which a participant sees the query in both SQL (left) and our QueryVis diagram (right).

have to specify anything upfront and can just copy the SQL query and the schema into the two available forms (notice that the relevant part of the schema could often be inferred from the query).

4.1 User study showing users can interpreting queries faster with QueryVis

We designed a user study to test whether our diagrams help users understand SQL queries *in less time* and *with fewer errors*, on average. The study design and analysis plan was preregistered before we started the experiment and gathered data. Details on the study are available in [61] and on OSF at https://osf.io/mycr2.

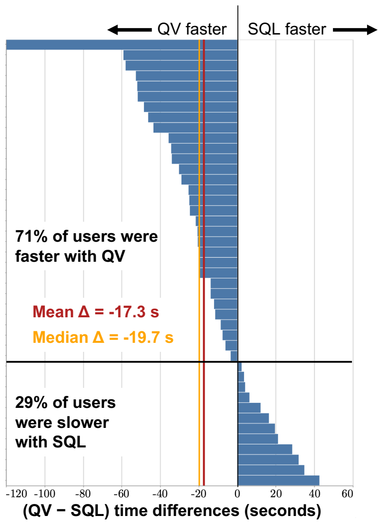
The study is an easily-scalable *within-subjects study* [84] (i.e., all study participants were exposed to all query interfaces). The study consisted of 9 multiple-choice questions (MCQs). Each MCQ asked the participant to choose the best interpretation for a presented query from 4 choices. Following best practices in MCQ creation [99], we created all 4 choices to read very similar to each other so that a participant with little knowledge of SQL would be incapable of eliminating any of the 4 choices. Each query was presented to participants in one of 3 conditions: (1) seeing a query as SQL alone (“SQL”), (2) seeing a query as a logical diagram that was generated from SQL (“QV”), or (3) seeing both SQL and QueryVis at the same time (“Both”). Figure 11 shows the interface for the condition “Both” for one of the 9 questions. Each participant answered *all 9 questions in the same order* but the condition for each question was randomized in a particular way that reduces potential biases in our analysis due to condition ordering effects following a *Latin square design* [59, 71]. We then tracked the time needed and errors made by each participant while trying to find the correct interpretation for each query. Participants had to pass an SQL qualification exam to ensure that they had at least a basic proficiency with SQL. We made the study available for 3 weeks from Jan 24, 2020–Feb 13, 2020 on Amazon Mechanical Turk (AMT), during which we recruited $n = 42$ legitimate participants.

Results. There is strong evidence that participants are meaningfully faster (-20%) using QueryVis than SQL ($p < 0.001$). There is weak evidence that participants make meaningfully fewer errors (-21%) using QueryVis than SQL ($p = 0.15$). Figure 12 shows the full time and error difference distribution across the 42 participants.

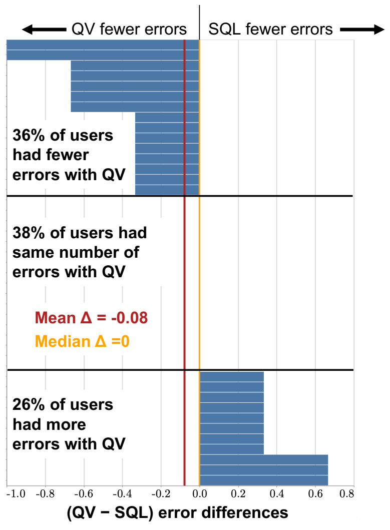
5 Related work

For decades, SQL has been the main standard for issuing queries over relational databases. There is a reasonable chance that this widely implemented interface won’t be replaced anytime soon. Thus we do not propose new ways to write queries, but instead explore how to help users *understand existing SQL queries*.

Visual Query Languages (VQL). Visual methods for expressing queries have been studied extensively in the database literature [12], and many commercial database products offer some visual interface for users to write



(a)



(b)

Figure 12: Section 4.1: Distribution of (QV−SQL) time and error differences for each participant on 9 MCQs.

simple SQL queries. Query Visualization (QV) focuses on the problem of describing and *interpreting a query that has already been written*, which is different from the problem of composing a new query (Section 2.1).

Interactive query builders employ visual diagrams that users can manipulate (most often in order to select tables and attributes) while using *a separate query configurator* (similar to QBE's condition boxes [100]) to specify selection predicates, attributes, and sometimes nesting between queries. dbForge [23] is the most advanced and commercially supported tool we found for interactive query building. Yet it does not show any visual indication for non-equi joins between tables and the actual filtering values and aggregation functions can only be added in a separate query configurator. Moreover, it has limited support for nested queries: the inner and outer queries are built separately, and the diagram for the inner query is *presented separately and disjointly* from the diagram for the outer query. Thus *no visual depiction of correlated subqueries is possible*. Other graphical SQL editors like SQL Server Management Studio (SSMS) [89], Active Query Builder [2], QueryScope from SQLdep [78], MS Access [68], and PostgreSQL's pgAdmin3 [76] lack in even more aspects of visual query representations: most do not allow nested queries, none has a single visual element for the logical quantifiers `NOT EXISTS` or `FOR ALL`, and all require specifying details of the query in SQL or across several tabbed views *separate from a visual diagram*. DataPlay [1] allows a user to specify their query by interactively modifying a *query tree with quantifiers* and observing changes in the matching/non-matching data. Gestural query specification [73] allows a user to query databases using a series of gestures on a touchscreen. In short, current graphical SQL editors *do not provide a single encompassing visualization of a query*. Thus they could not (even in theory) transform a complicated SQL query into a single visual representation, which is the focus of query visualization.

Query visualizations attempt to create succinct visual representations of existing queries. This explicit reverse functionality for SQL has not drawn as much attention as visual query builders, and there are only a handful of other systems we are aware of [32]. Visual SQL [48] is a visual query language that also support 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 (Figure 10). SQLVis [69] 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. Snowflake join [88] is an open source project that visualizes join queries with optional grouping. It does not have any consistent and unambiguous notation for nested queries. GraphSQL [13] uses visual metaphors that are different from typical relational schema notations and visualizations, even simple conjunctive queries can look unfamiliar. The Query Graph Model (QGM) developed for Starburst [40] helps users understand query plans, not query intent (Section 2.2). StreamTrace [8] focuses on visualizing temporal queries with workflow diagrams and a timeline. It is an example of visualizations for spatio-temporal domains and not the logic behind general relational queries.

6 Various Challenges

A vision paper that some of us wrote in 2011 declared that "Databases will visualize queries too" [31]. This has not yet widely happened. Why so? Is it just a matter of time? Or is it that we are missing something profound and Queries Visualizations (QV) will go the same route as Visual Query Languages (VQL): intuitively attractive, but practically not as useful? Instead, perhaps the foundations have been laid, yet there are still problems to be solved to make that vision practical. Here is a partial list of several such challenges:

(1) Extensions: Expressing logical disjunction in diagrams is inherently more complicated than conjunction [85]. And while relational algebra, relational calculus, and Datalog all use set semantics, SQL uses bag semantics. What are *appropriate visual metaphors* for general disjunctions and non-logical constructs, such as groupings, aggregates, arithmetic predicates, bag semantics, outer joins, null values, and recursion?[9]

(2) Relational patterns: Something not well understood or even formalized today is the vague concept of "relational query patterns." What are query patterns? We posit that identifying patterns in queries may have several advantages, akin to how formalizing best practices in software design patterns has aided software engineers [29]. General and reusable query patterns could assist in teaching students how to write complicated queries. Queries written using common patterns could then potentially be easier to interpret quickly. What is *a rigorous semantic definition of relational query patterns*? See [33] for first steps in that direction.

(3) Measures of visual conciseness: We listed minimal visual complexity as guiding principle. When comparing two languages one could likely develop metrics such as amount of ink used. However, what is the ultimately right measure for quantifying visual complexity for a human user? Visual complexity also needs to take into account prior familiar notions (like UML) to a target audience.

(4) Automatic layout algorithms: The online QueryVis demo uses a layered arrangement of tables that guarantees that any join conditions are either between two adjacent layers or within the same layer. It uses the standard Graphviz library for arranging the tables and their attributes [25]. However, existing graph layout algorithms are not suited for complicated layered graphs with nested hierarchies. What are new outline algorithms that can optimize for existing visual metrics (such as minimum line overlap) and define novel metrics that capture visual homogeneity? A possible route is encoding existing aesthetic heuristics (such as those in [9, Table 1]) or novel ones in quantitative metrics, and then defining the layout problem as Integer Linear Program (ILP). A recent proposal that uses this route is STRATISFIMAL LAYOUT [6].

(5) Interactive diagrams: As already implied by Figure 1, Figure 2, Figure 3,and Figure 9, the interaction between textual query and query visualization could be more involved beyond a simple one-way translation. Interactive mouse-over can show correspondences. An interactive auto-complete feature could suggest possible query templates. And a user could be allowed to manipulate an existing template of a query, which then gets reflected in the text (but notice that this last point would defeat the original idea to keep the visualization lightweight and as easy add-on to an existing query composition workflow). What is the optimal end-to-end integration of visualization and text or alternative input forms (recall Figure 3 and Figure 2)?

(6) Combinations with other modalities: We mentioned in Section 2.4 alternative ways to help users understand their queries. Such alternative modalities could possibly be combined with visualizations. For example, Natural Language translations could possibly benefit from a graphical representation of a query. Parts of a query could be replaced with an automatically created text and expanded with a click to the full pattern. Query visualizations could be enhanced with example database instances, or operator-by-operator translations. The

[9]The online QueryVis interface at https://queryVis.com has been quietly supporting limited forms of aggregates already since 2016.

query visualizations could be modified to display how individual records fit or don't fit the query. This could again be done with interactive mouseover or choice from a menu.

(7) User studies: We believe that preregistered, within-subjects studies with multiple-choice questions in Latin square design studies, where the correctness of a user's answer can be determined automatically, are the way to go to for easily scalable quantitative comparison between different interfaces. In our user study, users needed to pass a SQL qualifying exam and then started the test only after minimum exposure to QueryVis. One could only imagine the improvement after the users had chance to become more familiar with the visual language over an extended period of time. For such a longitudinal study that possibly instruments across control groups there is one big open challenge: How to parameterize SQL queries and questions in the spirit of Gradiance [35] such that the same subjects can take the same test repeatedly? Such new user study paradigms would allow us to observe the improvement in speed and accuracy over an extended period of time.

(8) Declarative programming: Logical query interfaces have shown success also beyond relational data. Examples include Datalog for networks [46, 63, 64] and Inductive Logic Programming [83]. Such programs represent an explicit symbolic structure that can be inspected and understood, and the implied logic visualized.

7 Conclusions and Future Work

We discussed the potential of query visualizations for a future, advanced user-query interaction that visualizes relational patterns of a query with diagrams. We delineated query visualization from visual query languages, discussed a few principles for designing intuitive visual diagrams, and gave two variants of a family of query visualizations. Future work needs to extend visual formalisms for the full relational model and beyond, find algorithms for automatic arrangement of diagrams, study novel and more interactive user interfaces with query visualizations being just one component, create more easily verifiable, large-scale user studies, and find ways to apply logical representations to other logic-based languages such as Inductive Logic Programming.

Acknowledgements

This research is supported in part by NSF awards IIS-1762268, IIS-1956096, and IIS-2145382. WG would also like to thank Jonathan Danaparamita who created the original interactive QueryVis demo that has been online at https://queryvis.com/ since 2011 and without whom there would have been no QueryVis.

References

[1] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Dataplay: interactive tweaking and example-driven correction of graphical database queries. In *Proceedings of the 25th annual ACM symposium on User interface software and technology (UIST)*, pages 207–218, 2012. https://doi.org/10.1145/2380116.2380144.

[2] Active Query Builder. https://www.activequerybuilder.com/, 2019.

[3] T. M. Ahmed, W. Shang, and A. E. Hassan. An empirical study of the copy and paste behavior during development. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 99–110, 2015. https://doi.org/10.1109/MSR.2015.17.

[4] J. Akbarnejad, G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, D. On, N. Polyzotis, and J. S. V. Varman. SQL QueRIE recommendations. *PVLDB*, 3(1):1597–1600, 2010. https://doi.org/10.1145/2839509.2844640.

[5] N. Arzamasova and K. Böhm. Scalable and data-aware sql query recommendations. *Information Systems*, 96:101646, 2021. https://doi.org/10.1016/j.is.2020.101646.

[6] S. D. Bartolomeo, M. Riedewald, W. Gatterbauer, and C. Dunne. STRATISFIMAL LAYOUT: A modular optimization model for laying out layered node-link network visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):324–334, 2022. https://doi.org/10.1109/TVCG.2021.3114756, https://visdunneright.github.io/stratisfimal/.

[7] L. Battle. Analyzing online programming communities to enhance visualization languages. *Interactions*, 29(1):27–29, jan 2022. https://doi.org/10.1145/3503490.

[8] L. Battle, D. Fisher, R. DeLine, M. Barnett, B. Chandramouli, and J. Goldstein. Making sense of temporal queries with interactive visualization. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5433–5443, 2016. https://doi.org/10.1145/2858036.2858408.

[9] C. Bennett, J. Ryall, L. Spalteholz, and A. Gooch. The aesthetics of graph visualization. In *3rd International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging (CompAesth)*, pages 57–64. Eurographics Association, 2007. https://doi.org/10.2312/COMPAESTH/COMPAESTH07/057-064.

[10] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Writing code to prototype, ideate, and discover. *IEEE Software*, 26(5):18–24, 2009. https://doi.org/10.1109/MS.2009.147.

[11] T. Catarci. Visual query language. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. https://doi.org/10.1007/978-1-4614-8265-9_448.

[12] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8(2):215–260, 1997. https://doi.org/10.1006/jvlc.1997.0037.

[13] C. Cerullo and M. Porta. A system for database visual querying and query visualization: Complementing text and graphics to increase expressiveness. In *International Workshop on Database and Expert Systems Applications (DEXA)*, pages 109–113. IEEE, 2007. https://doi.org/10.1109/DEXA.2007.91.

[14] H. C. Chan, K. K. Wei, and K. L. Siau. User-database interface: The effect of abstraction levels on query performance. *MIS Quarterly*, 17(4):441–464, 1993. https://doi.org/10.2307/249587.

[15] B. Chandra, B. Chawda, B. Kar, K. V. M. Reddy, S. Shah, and S. Sudarshan. Data generation for testing and grading SQL queries. *VLDB J.*, 24(6):731–755, 2015. https://doi.org/10.1007/s00778-015-0395-0.

[16] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Query recommendations for interactive database exploration. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, volume 5566 of *LNCS*, pages 3–18. Springer, 2009. https://doi.org/10.1007/978-3-642-02279-1_2.

[17] C. Chen. *Information visualization: beyond the horizon*. Springer, New York, 2nd edition, 2006. https://doi.org/10.1007/1-84628-579-8.

[18] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for SQL. In *8th biennial Conference on Innovative Data Systems Research (CIDR)*, 2017. http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf.

[19] A. Ciborowska, N. A. Kraft, and K. Damevski. Detecting and characterizing developer behavior following opportunistic reuse of code snippets from the Web. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 94–97. ACM, 2018. https://doi.org/10.1145/3196398.3196467.

[20] G. J. Clark and C. T. Wu. DFQL: Dataflow query language for relational databases. *Information & Management*, 27(1):1–15, 1994. https://doi.org/10.1016/0378-7206(94)90098-1.

[21] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, USA, 2nd edition, 2006. https://doi.org/10.1002/047174882X.

[22] J. Danaparamita and W. Gatterbauer. Queryviz: Helping users understand SQL queries and their patterns. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT)*, pages 558–561. ACM, 2011. https://doi.org/10.1145/1951365.1951440, https://queryvis.com/.

[23] dbForge. https://www.devart.com/dbforge/mysql/querybuilder/, 2019.

[24] M. Eirinaki, S. Abraham, N. Polyzotis, and N. Shaikh. QueRIE: Collaborative database exploration. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26(7):1778–1790, 2014. https://doi.org/10.1109/TKDE.2013.79.

[25] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull. *Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools*, pages 127–148. Springer, 2004. https://doi.org/10.1007/978-3-642-18638-7_6.

[26] J. Fan, G. Li, and L. Zhou. Interactive SQL query suggestion: Making databases user-friendly. In *27th International Conference on Data Engineering (ICDE)*, pages 351–362, 2011. https://doi.org/10.1109/ICDE.2011.5767843.

[27] A. Fish and J. Howse. Towards a default reading for constraint diagrams. In *International Conference on Theory and Application of Diagrams (DIAGRAMS)*, pages 51–65. Springer, 2004. https://doi.org/10.1007/978-3-540-25931-2_8.

[28] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman, 3rd edition, 2003. https://dl.acm.org/doi/10.5555/861282.

[29] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995. https://dl.acm.org/doi/book/10.5555/186897.

[30] S. Gathani, P. Lim, and L. Battle. Debugging database queries: A survey of tools, techniques, and users. In *Conference on Human Factors in Computing Systems (CHI)*, pages 1–16, 2020. https://doi.org/10.1145/3313831.3376485.

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

[32] W. Gatterbauer. Interpreting and understanding relational database queries using diagrams. International Conference on Theory and Application of Diagrams (DIAGRAMS) – Tutorials, 2022. http://www.diagrams-conference.org/2022/index.php/program/tutorials/.

[33] W. Gatterbauer, C. Dunne, and M. Riedewald. Relational diagrams: a pattern-preserving diagrammatic representation of non-disjunctive relational queries. *Arxiv preprint arXiv:2203.07284*, 2022. https://arxiv.org/abs/2203.07284.

[34] S. Gehrmann, F. Z. Dai, H. Elder, and A. M. Rush. End-to-end content and plan selection for data-to-text generation. In *International Conference on Natural Language Generation (INLG)*, pages 46–56. ACM, 2018. https://doi.org/10.18653/v1/w18-6505.

[35] Gradiance. https://www.gradiance.com/services, 2022.

[36] S. L. Greene, S. J. Devlin, P. E. Cannata, and L. M. Gomez. No IFs, ANDs, or ORs: A study of database querying. *International Journal of Man-Machine Studies*, 32(3):303–326, 1990. https://doi.org/10.1016/S0020-7373(08)80005-3.

[37] T. Grust, F. Kliebhan, J. Rittinger, and T. Schreiber. True language-level SQL debugging. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT)*, pages 562–565, 2011. https://doi.org/10.1145/1951365.1951441.

[38] T. Grust and J. Rittinger. Observing SQL queries in their natural habitat. *ACM Transactions on Database Systems (TODS)*, 38(1):3, 2013. https://doi.org/10.1145/2445583.2445586.

[39] P. Guagliardo and L. Libkin. Correctness of SQL queries on databases with nulls. *SIGMOD Record*, 46(3):5–16, 2017. https://doi.org/10.1145/3156655.3156657.

[40] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. *SIGMOD Record*, 18(2):377–388, 1989. https://doi.org/10.1145/67544.66962.

[41] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, and V. Vianu. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001. https://doi.org/10.2307/2687775.

[42] E. C. Harel and E. R. McLean. The effects of using a nonprocedural computer language on programmer productivity. *MIS Quarterly*, 9(2):109–120, jun 1985. https://doi.org/10.2307/249112.

[43] J. R. Haritsa. The Picasso database query optimizer visualizer. *PVLDB*, 3(2):1517–1520, 2010. https://doi.org/10.14778/1920841.1921027.

[44] B. Howe and G. Cole. SQL is dead; long live SQL: Lightweight query services for ad hoc research data. In *4th Microsoft eScience Workshop*, 2010. https://homes.cs.washington.edu/~billhowe/projects/2014/03/22/SQLShare.html.

[45] J. Howse. Diagrammatic reasoning systems. In *International Conference on Conceptual Structures (ICCS)*, volume 5113 of *LNCS*, pages 1–20. Springer, 2008. https://doi.org/10.1007/978-3-540-70596-3_1.

[46] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1213–1216, 2011. https://doi.org/10.1145/1989323.1989456.

[47] Y. E. Ioannidis. From databases to natural language: The unusual direction. In *International Conference on Application of Natural Language to Information Systems (NLDB)*, volume 5039 of *LNCS*, pages 12–16. Springer, 2008. https://doi.org/10.1007/978-3-540-69858-6_3.

[48] H. Jaakkola and B. Thalheim. Visual sql – high-quality er-based query treatment. In *Workshops @ International Conference on Conceptual Modeling (ER)*, LNCS, pages 129–139. Springer, 2003. https://doi.org/10.1007/978-3-540-39597-3_13.

[49] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. SQLShare: Results from a multi-year SQL-as-a-service experiment. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 281–293, 2016. https://doi.org/10.1145/2882903.2882957.

[50] M. Jarke, J. Tuner, E. Stohr, Y. Vassiliou, N. White, and K. Michielsen. A field evaluation of natural language for data retrieval. *IEEE Transactions on Software Engineering*, SE-11(1):97–114, 1985. https://doi.org/10.1109/TSE.1985.231847.

[51] M. Jarke and Y. Vassiliou. A framework for choosing a database query language. *ACM Computing Surveys*, 17(3):313–340, 1985. https://doi.org/10.1145/5505.5506.

[52] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 238–247, 2008. https://doi.org/10.1109/ASE.2008.34.

[53] M. A. Khan, L. Xu, A. Nandi, and J. M. Hellerstein. Data tweening: Incremental visualization of data transforms. *PVLDB*, 10(6):661–672, 2017. https://doi.org/10.14778/3055330.3055333.

[54] N. Khoussainova, M. Balazinska, W. Gatterbauer, Y. Kwon, and D. Suciu. A case for a collaborative query management system. In *4th biennial Conference on Innovative Data Systems Research (CIDR)*, 2009. https://database.cs.wisc.edu/cidr/cidr2009/Paper_94.pdf.

[55] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: A context-aware SQL-autocomplete system. *PVLDB*, 4(1):22–33, 2010. https://doi.org/10.14778/1880172.1880175.

[56] G. Koutrika, A. Simitsis, and Y. E. Ioannidis. Explaining structured queries in natural language. In *International Conference on Data Engineering (ICDE)*, pages 333–344. IEEE, 2010. http://doi.org/10.1109/ICDE.2010.5447824.

[57] LaTeX. https://en.wikipedia.org/wiki/LaTeX, 2022.

[58] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 492–501, 2006. https://doi.org/10.1145/1134285.1134355.

[59] J. Ledolter and A. J. Swersey. *Testing 1-2-3: experimental design with applications in marketing and service operations*. Stanford Business Books, 2007. https://www.sup.org/books/title/?id=4513.

[60] J. Leggett and G. Williams. An empirical investigation of voice as an input modality for computer programming. *International Journal of Man-Machine Studies*, 21(6):493–520, 1984. https://doi.org/10.1016/S0020-7373(84)80057-7.

[61] A. Leventidis, J. Zhang, C. Dunne, W. Gatterbauer, H. V. Jagadish, and M. Riedewald. Queryvis: Logic-based diagrams help users understand complicated SQL queries faster. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2303–2318, 2020. https://doi.org/10.1145/3318464.3389767, Full version: https://osf.io/btszh/.

[62] G. Li, J. Fan, H. Wu, J. Wang, and J. Feng. DBease: Making databases user-friendly and easily accessible. In *5th biennial Conference on Innovative Data Systems Research (CIDR)*, pages 45–56, 2011. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper6.pdf.

[63] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009. https://doi.org/10.1145/1592761.1592785.

[64] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 75–90, 2005. https://doi.org/10.1145/1095810.1095818.

[65] P. Marcel and E. Negre. A survey of query recommendation techniques for datawarehouse exploration. In *7th Conference on Data Warehousing and On-Line Analysis (EDA)*, 2011. http://eda2011.cemagref.fr/.

[66] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. WHY so? or WHY no? Functional causality for explaining query answers. In *Proceedings of the 4th International VLDB workshop on Management of Uncertain Data (MUD)*, pages 3–17, 2010. http://arxiv.org/abs/0912.5340.

[67] Z. Miao, S. Roy, and J. Yang. Explaining wrong queries using small examples. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 503–520, 2019. https://doi.org/10.1145/3299869.3319866.

[68] Microsoft Access. https://products.office.com/en-us/access, 2019.

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

[70] T. Milo and A. Somech. React: Context-sensitive recommendations for data analysis. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 2137–2140, 2016. https://doi.org/10.1145/2882903.2899392.

[71] D. C. Montgomery. *Design and analysis of experiments*. John Wiley & Sons, Inc., 8th edition, 2013. https://dl.acm.org/doi/book/10.5555/1206386.

[72] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97 – 123, 1990. https://doi.org/10.1016/S1045-926X(05)80036-9.

[73] A. Nandi, L. Jiang, and M. Mandel. Gestural query specification. *PVLDB*, 7(4):289–300, 2013. https://doi.org/10.14778/2732240.2732247.

[74] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 245–256, 2009. https://doi.org/10.1145/1559845.1559873.

[75] C. S. Peirce. Charles Hartshorne and Paul Weiss (Editors). Collected papers of Charles Sanders Peirce. vol. 4. *The ANNALS of the American Academy of Political and Social Science*, 1933. https://doi.org/10.1177/000271623417400185.

[76] pgAdmin. https://www.pgadmin.org/, 2019.

[77] PostgreSQL. https://www.postgresql.org/, 2022.

[78] QueryScope. https://sqldep.com/, 2019.

[79] P. Reisner. Human factors studies of database query languages: A survey and assessment. *ACM Computing Surveys*, 13(1):13–31, 1981. https://doi.org/10.1145/356835.356837.

[80] P. Reisner, R. F. Boyce, and D. D. Chamberlin. Human factors evaluation of two data base query languages: Square and sequel. In *Proceedings of the May 19-22, 1975, national computer conference and exposition (AFIPS)*, pages 447–452. ACM, 1975. https:/doi.org/10.1145/1499949.1500036.

[81] S. P. Reiss. Visual representations of executing programs. *Journal of Visual Languages & Computing*, 18(2):126–148, 2007. https://doi.org/10.1016/j.jvlc.2007.01.003.

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

[83] U. Schmid, C. Zeller, T. Besold, A. Tamaddoni-Nezhad, and S. Muggleton. How does predicate invention affect human comprehensibility? In *International Conference on Inductive Logic Programming (ILP)*, volume 10326 of *LNCS*, pages 52–67. Springer, 2017. https://doi.org/10.1007/978-3-319-63342-8_5.

[84] H. J. Seltman. *Experimental design and analysis*. Carnegie Mellon University, 2012. http://www.stat.cmu.edu/~hseltman/309/Book/Book.pdf.

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

[86] A. Simitsis and Y. E. Ioannidis. DBMSs should talk back too. In *4th biennial Conference on Innovative Data Systems Research (CIDR)*, 2009. https://database.cs.wisc.edu/cidr/cidr2009/Paper_119.pdf.

[87] Sloan Digital Sky Survey (SDSS), 2022. https://www.sdss.org/.

[88] Snowflake join. http://revj.sourceforge.net, 2022.

[89] SQL Server Management Studio. https://www.microsoft.com/en-us/sql-server/sql-server-downloads, 2019.

[90] sqlparse. https://pypi.org/project/sqlparse/, 2022.

[91] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001. https://www.edwardtufte.com/tufte/books_vdqi.

[92] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning (LPAR)*, volume 6355 of *LNCS*, pages 425–446. Springer, 2010. https://doi.org/10.1007/978-3-642-17511-4_24.

[93] W. Wang, S. S. Bhowmick, H. Li, S. R. Joty, S. Liu, and P. Chen. Towards enhancing database education: Natural language generation meets query execution plans. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, pages 1933–1945, 2021. https://doi.org/10.1145/3448016.3452822.

[94] C. Welty and D. W. Stemple. Human factors comparison of a procedural and a nonprocedural query language. *ACM Transactions on Database Systems (TODS)*, 6(4):626–649, 1981. https://doi.org/10.1145/319628.319656.

[95] K. Xu, L. Wu, Z. Wang, Y. Feng, and V. Sheinin. SQL-to-text generation with graph-to-sequence model. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 931–936. ACL, 2018. https://doi.org/10.18653/v1/d18-1112.

[96] D. Yang, P. Martins, V. Saini, and C. Lopes. Stack overflow in Github: Any snippets there? In *14th International Conference on Mining Software Repositories (MSR)*, pages 280–290, 2017. https://doi.org/10.1109/MSR.2017.13.

[97] M.-M. Yen and R. Scamell. A human factors experimental comparison of SQL and QBE. *IEEE Transactions on Software Engineering*, 19(4):390–409, 1993. https://doi.org/10.1109/32.223806.

[98] K. Zhang. *Visual languages and applications*. Springer, New York, 2007. https://doi.org/10.1007/978-0-387-68257-0.

[99] D. M. Zimmaro. *Writing Good Multiple-Choice Exams*. Center for Teaching and Learning, UT Austin., 2010. https://facultyinnovate.utexas.edu/sites/default/files/writing-good-multiple-choice-exams-fic-120116.pdf.

[100] M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977. https://doi.org/10.1147/sj.164.0324.