Bulletin of the Technical Committee on

Data Engineering

September 2022 Vol. 45 No. 3

IEEE Computer Society

Letters

Letter from the Editor-in-Chief . . . . . *Haixun Wang* 1
Letter from the Special Issue Editors . . . . . *Sudeepa Roy and Jun Yang* 2

Special Issue on Widening the Impact of Data Engineering through Innovations in Education, Interfaces, and Features

Automated Grading of SQL Queries . . . . . *Bikash Chandra and S. Sudarshan* 4
Towards Technology-Enabled Learning of Relational Query Processing . . . . . *Sourav S. Bhowmick and Hui Li* 16
Principles of Query Visualization . . . . *Wolfgang Gatterbauer, Cody Dunne, H.V. Jagadish, and Mirek Riedewald* 34
Structured Data Representation in Natural Language Interfaces . . *Yutong Shao, Arun Kumar, and Ndapa Nakashole* 55
Quill: A Declarative Approach for Accelerating Augmented Reality Application Development . . . . . *Codi Burley, Ritesh Sarkhel, and Arnab Nandi* 69
In-Database Decision Support: Opportunities and Challenges . . *Azza Abouzied, Peter J. Haas, and Alexandra Meliou* 89
User Interfaces for Exploratory Data Analysis: A Survey of Open-Source and Commercial Tools . . . . . *Jinglin Peng, Weiyuan Wu, Jing Nathan Yan, Danrui Qi, Jeffrey M. Rzeszotarski, and Jiannan Wang* 103
The Right Tool for the Job: Data-Centric Workflows in Vizier . . . . . *Oliver Kennedy, Boris Glavic, Juliana Freire, and Michael Brachmann* 116

Conference and Journal Notices

TCDE Membership Form 132

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

The current issue of the Bulletin is about tools and education for data management practitioners. The first paper in the issue

Haixun Wang
Instacart

Letter from the Special Issue Editors

The longevity of the database systems is truly something to marvel at: it has been more than half a century since the introduction of the relational model in 1970. The increasingly important role played by data in modern-day endeavors has further contributed to growing interest in data management. More students—from not only computer science and data science but also other disciplines—see data management skills as an essential part of their training. Nonetheless, this is no time for complacence by the data engineering community. First, our methods for teaching data management skills are becoming woefully inadequate in the presence of growing number of students and diversity in their backgrounds. Second, the exploding number and variety of new data-driven applications demand constant innovations in data management that go far beyond the interfaces and features of traditional database systems. This special issue is devoted to sampling ongoing research from our community to address the above challenges.

The issue begins with two projects focused on improving teaching and learning relational database technology. **Chandra and Sudarshan** present the work on the *XData* system at IIT Bombay, which automatically grades student queries given a set of correct queries. XData generates multiple datasets to catch different errors ensuring better test coverage, it suggests multiple correct query structures that are helpful for partial grading, and also provides individualized feedback on what changes are needed to correct a query. **Bhowmick and Li** describe the *TRUSS* project at the Nanyang Technological University, which helps students learn relational query processing. Its goals include helping users understand executing plans, as well as how database optimizers choose among various alternative plans. Guided by the motivation theories of learning and informed by data collected throughout the learning process, *TRUSS* employs a variety of modes to assist learners, such as natural language explanations, visualizations, and an interactive chatbot. Both papers share their experience of using these two systems in database courses at their respective universities.

The third paper, by **Gatterbauer et al.**, presents the principles of query visualization, which aims at helping humans understand the meaning of a query written in SQL. This visualization paradigm has not only applications in education, but with a graphical representation of queries that abstract away unnecessary syntactical details, it also enables interesting uses such as identifying code reuse opportunities and clustering query patterns.

The subsequent papers in this issue seek to extend database technology for novel data interfaces and applications. The paper from UC San Diego by **Shao et al.** tackles a key challenge in building natural language interfaces for data: how to learn representations of structured data, for text- and speech-to-SQL translation, as well as for building dialog systems for tasks such as helping users produce plots from structure data. The paper from Ohio State by **Burley et al.** is about support for developing augmented reality applications. These applications virtually augment real-world objects with additional information by performing real-time "joins" between data extracted from the physical environment with data from remote sources. The *Quill* framework they present simplifies the development process using declarative specifications, and uses automatic optimization to achieve better performance than code developed with standard tools. The paper from NYU Abu Dhabi and UMass Amherst by **Abouzied et al.** discusses opportunities for in-database decision support and the challenges related to usability, scalability, data uncertainty, dynamic environments with changing data and models, and robust policy making. Their *PackageBuilder* system and its extensions transform a declarative specification of a package query into an integer linear program, and solve it on deterministic or uncertain data to obtain the desired package of input tuples with minimum cost or maximum profit while satisfying specified constraints.

The last two papers in this issue are about exploratory data analysis (EDA). **Peng et al.** present a survey of several open-source and commercial interfaces for exploratory data analysis. They focus on user exploration activities in three stages of EDA: generating initial questions to get some initial questions to explore, creating visualizations to answer questions, and examining the visualizations to produce answers or ask more questions. **Kennedy et al.** describe *Vizier*, an extensible multi-modal platform for data-centric workflows. The main idea behind Vizier is that, rather than alternating between task-specific tools such as spreadsheets and notebooks for data exploration, discovery, and analysis, a better approach is to build multiple user interfaces on top of a single

incremental workflow/dataflow platform with built-in support for versioning, provenance, error discovery, output tracking, and data cleaning.

Overall, we hope these eight articles together offer a sample of the ongoing work as well as exciting challenges in widening the impact of database engineering community — both through education and through novel interfaces and features. We would like to thank the authors of this issue for their contributions, and welcome more from our community to join them.

Sudeepa Roy and Jun Yang
Duke University, USA

Automated Grading of SQL Queries

Bikash Chandra* S. Sudarshan
IIT Bombay
{bikash,sudarsha}@cse.iitb.ac.in

Abstract

In a traditional classroom setting, instructors and teaching assistants either grade SQL queries either manually or using fixed datasets. Manual grading is tedious, error-prone and does not scale well for courses with a large number of students or for online courses. Using fixed datasets may miss even common errors and mark incorrect student queries as correct. In this article, we discuss our system XData that, given a set of correct queries, can automatically evaluate the correctness of SQL queries using datasets designed to catch errors in the given correct queries. If a student query is found to be incorrect, XData can even assign grades to the query based on how close they are to a correct query. In our experience, these techniques can be used to grade queries for settings with a large number of students with little human effort and involvement.

1 Introduction

Complex SQL queries may be written in several different ways and are difficult for beginners to get right. In courses that teach SQL queries, student SQL queries are often graded manually. The manual grading involves reading the student query manually comparing it to a correct query and/or executing the query on fixed datasets. Manually reading the query and comparing queries may be difficult and is also prone to errors while using grading using fixed datasets may miss errors in student queries. Let us consider an example where the correct query Q is

```
SELECT course.id, department.dept_name
FROM course LEFT OUTER JOIN
  (SELECT * from department WHERE department.budget > 70000) d
USING (dept_name);
```

A common mistake made by students is to write the following query Q_s instead

```
SELECT course.id, department.dept_name
FROM course LEFT OUTER JOIN department
USING (dept_name) WHERE department.budget > 70000;
```

The student query looks sufficiently similar for a grader to miss the difference. The queries, however, are not equivalent since they give different results on departments with a budget less than 70000. The query Q would output such courses with a NULL department name while Q_s would not output such courses. Even using a fixed

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Currently at Meta Platforms Inc.

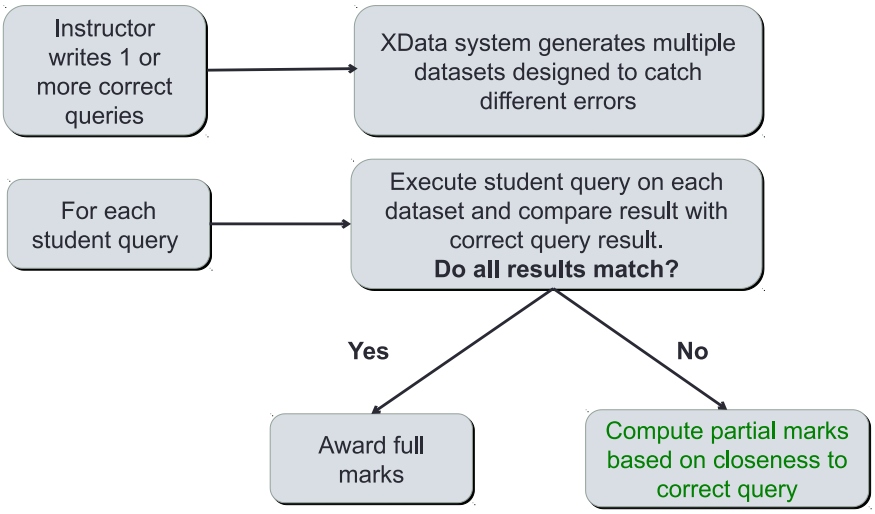


Figure 1: Automated Grading Workflow

dataset may not be able to find the difference unless the dataset has a tuple where the department budget is less than 70000. Thus a fixed dataset may also miss such errors in grading.

Even when the student query is incorrect, the grader is often expected to provide partial marks to the student query based on how close the student query is to being correct. A naive approach of counting the number of datasets for which a query gives the correct answer may not be fair for partial marking since a small error may cause many or all test datasets to fail. For example, if a student used a selection condition $a > 10$ instead of $a < 10$, most test datasets would fail. Conversely, a query with basic conceptual errors may still give the correct result on some datasets, especially ones where an empty answer is expected. Awarding partial marks manually is tedious and error-prone. Let us consider the following correct query provided by the instructor

```
SELECT * FROM r INNER JOIN s ON (r.A=s.A) WHERE r.A>10
```

and a student submitted the query

```
SELECT * FROM r INNER JOIN s ON (r.A=s.B) WHERE s.A>10
```

A grader manually evaluating the student query above may deduct marks for two errors - one for the join condition and another for the selection condition. However, if the join condition in the student query is fixed, the student query is equivalent to the given correct query since now r.A and s.A are equivalent in the student query. Hence only marks for one error should have been deducted.

In a database course, it would also be helpful for the student to receive specific feedback as to where they went wrong and how their mistakes could have been corrected. This is very time-consuming for TAs and is rarely done well even for moderately sized classes. With the growing popularity of online courses, where students expect instant feedback on the answers they submit there is an increasing need for automated and instantaneous feedback, Manual grading does not scale for large online courses, and just showing datasets where the query gave a wrong result may not provide clear feedback to students.

In our work on XData [1–3], we developed techniques to automatically grade student queries, given a set of correct queries. XData has two steps for grading student SQL queries as shown in Figure 1. The instructor first provides the question text and some correct queries. Based on the correct queries, XData generates multiple datasets that are tailored to catch different types of errors on the given queries. Since SQL queries may be written in several different ways, allowing instructors to specify multiple correct queries allows us to ensure more coverage of test cases. It also helps us get more query correct structures which is useful for our partial marking technique. The test data generation technique can also be used to test database queries and applications as described in [1, 4].

When evaluating a student query, the student query is run against the datasets generated by XData and the results are compared with the correct query. For correct student queries, the results generated by the student query and the instructor query would be the same across all generated datasets. Such queries would be awarded full marks. We note that techniques for checking query equivalence could potentially be used to check for equivalence of a student query to a correct query, but the state of the art for equivalence checking does not handle many SQL features such as null values, and has limitations in reasoning about equivalence with the given database constraints. While there is a risk of labeling an incorrect query as correct using our approach, we have not found it to be an issue in practice.

For incorrect queries, XData compares the student query with the correct query using an edit-based technique and provides a score as well as the changes that need to be made in the student query to make it a correct query. Our approach scales to large class sizes and can grade student queries and provide feedback instantly.

In this article, we first discuss, in Section 1, techniques for automatically generating test data and how the test data generated can be used to check the correctness of SQL queries submitted by students. In Section 3, we show how edit-based grading can be used to both award partial marks and provide individualized feedback for incorrect student queries. We share our experience of using the XData grading system in Section 4 and discuss related work in Section 5. We conclude the article in Section 6 and discuss some open challenges.

2 Using Datasets to Check Correctness

Test data generation in XData is done based on the correct queries provided by the instructor. Unlike fixed datasets that are query agnostic, these datasets are designed to catch errors based on the correct queries provided by the instructor and are hence much better at catching errors.

2.1 Common Errors in SQL Queries

Students make several types of errors when writing SQL queries. Some students may use an inner join when a left outer join was required, others may use a count aggregation when a count distinct was needed. Mutation testing is a well-known approach to check the adequacy of test cases for a program [5]. We use a similar approach for mutation testing of SQL queries. We consider mutations as (syntactically correct) changes to an SQL query. Errors made by student queries may be seen as mutations of the correct query and the incorrect query is called a non-equivalent mutant of the correct query. A dataset that produces different results on the correct and incorrect queries is said to kill the mutation.

XData produces multiple datasets. The first dataset is aimed to produce a non-empty result for the query. This dataset itself kills several mutations. For the remaining datasets, XData considers single mutations at a time on the correct query provided by the instructor and generates datasets that are targeted to kill the mutations. Each dataset is marked with a tag to indicate which type of mutations a dataset is designed to catch. Note that a dataset designed to catch one type of mutation may catch other types of mutations as well.

XData considers a large number of mutations in SQL queries including but not limited to the following.

- **Join type mutation**: A join type mutation involves replacing one of `{INNER, LEFT OUTER, RIGHT OUTER} JOIN` with another. Since the same join query may be written using different join orders, XData considers mutations across different join orders. Mutations involving missing or additional join orders are also considered.

- **Selection predicate mutation**: For selection conditions, XData considers mutations of the relational operator where any occurrence of one of $\{=, <>, <, >, \leq, \geq\}$ is replaced by another or if the selection condition is missing. XData also considers mutations of selection predicates between `IS NULL` and `NOT IS NULL` and for missing `IS NULL`. These predicates may be on integer, floating, text attributes or even aggregates. Mutations involving changing the constant in the selection condition are also considered.

- **Aggregation mutation**: Aggregations may be either unconstrained (at the root of the query tree) or constrained (having a condition with the aggregate). In both cases, the aggregation function can be mutated among `MAX, MIN, SUM, AVG, COUNT` and their `DISTINCT` versions. Mutations involving `COUNT(attr)` to `COUNT(*)`, in case `attr` is nullable, are also considered.

- **Group by attribute mutation**: For queries involving the `GROUP BY` clause, XData considers mutations involving additional or missing group by attributes both in the presence and absence of the `HAVING` clause.

- **Like operator mutation**: Like operators are used in SQL to match patterns in text attributes. SQL like operators include `LIKE, NOT LIKE, ILIKE and NOT ILIKE`. XData considers mutation of any one SQL like operator to other like operators. XData also considers mutations in the patterns used with the LIKE operator such as replacing as '%' with '_' and vice versa or missing '%' or '_' in the pattern.

- **Nested subquery mutations**: XData considers mutations between `IN` $vs.$ `NOT IN`, `EXISTS` $vs.$ `NOT EXISTS` and `ALL` $vs.$ `ANY/SOME`. Mutations on the queries in the nested subquery are also considered for test data generation so that errors inside subqueries are also caught.

- **Set operator mutations**: Set operators are used in compound queries to combine the results of two underlying results. Set operator mutations include changing one of the following operators to another: `UNION, UNION ALL, INTERSECT, INTERSECT ALL, EXCEPT, EXCEPT ALL`. Similar to nested subqueries, mutations of the subqueries whose results are input to these set operators are also considered.

- **Distinct mutation**: Duplicates in the results may be filtered using the `DISTINCT` clause. XData considers mutations of a missing or extraneous `DISTINCT` clause.

2.2 Test Data Generation to Detect Errors

For each type of mutation, we design specific conditions that the datasets must satisfy in order to kill such mutations. Let us take the following example query to demonstrate the mutations considered for queries with join and selections and how we generate datasets to kill the mutations.

```
    SELECT course.course_id
FROM course INNER JOIN takes USING(course_id)
WHERE course.credits >= 6
```

Some of the mutations that XData would consider and the techniques to kill those are the following.

1. **Join type mutation**: Consider the mutation from department INNER JOIN course to department LEFT OUTER JOIN course. In order to kill this mutation, we need to ensure that there is a tuple in department relation that does not satisfy the join condition with any tuple in course relation. The INNER JOIN query would not output that tuple in the department relation while the LEFT OUTER JOIN would.

   In general, a join query can be specified in a join order independent fashion, with many equivalent join orders for a given query. Hence, the number of join type mutations across all these orders is exponential. From the join conditions specified in the query, XData forms equivalence classes of <relation, attribute> pairs such that elements in the same equivalence class need to be assigned the same value to meet (one or more) join conditions. Using these equivalence classes, XData generates a linear number of datasets to kill join type mutations across all join orderings. If a pair of relations involve multiple join conditions

2. **Selection Predicate mutation**: For killing mutations for the selection condition A1 relop A2, XData generates 3 datasets where tuples satisfy the conditions (1) A1 $>$ A2, (2) A1 $<$ A2, and (3) A1 $=$ A2. These three datasets kill all non- equivalent mutations from one relop to another relop. These datasets also kill mutations because of missing selection conditions.

For the given query example the constraints would be (1) `course.credits>6`, (2) `course.credits<6` and (3) `course.credits=6`. If the student query uses a different operator instead of $>=$ or misses the selection condition one of the three datasets will catch the error.

Details on test data generation for killing other types of mutations are presented in [1]. We omit the details here for brevity.

It is not sufficient for only the conditions for killing mutations to be satisfied when generating the test database. The difference at one level, say the join condition must change the result of the query for the difference to be observed in the query result. For the given example query, consider the join mutation. If all tuples in course have grade less than 6, both the `INNER JOIN` and the `LEFT OUTER JOIN` would give empty results. Hence, when generating a dataset, XData ensures that the tuple has `grade >= 6`.

In order to generate a dataset, we generate constraints using an SMT solver [6]. In XData, we support CVC3 [7], CVC4 [8] as well as Z3 [9] as the constraint solvers. We encode text attributes as enumerates types and enumerates types are modeled as subtypes of integers or rationals. A tuple type is created for each relation to represent one row and an array of tuples represents the relation table. We also add other database constraints such as primary key and foreign key constraints, unique attribute constraints as well as domain constraints. The domain constraints ensure that we generate the correct types of values for each column of the database, the values generated are within the range specified by the schema and only nullable columns can have NULL values.

We also note that some mutations may be semantically equivalent to the correct queries and it may not be possible to kill such mutations. For such cases, the constraints for killing the mutations would not be satisfiable and the SMT solver would fail to generate the dataset to kill the mutation.

For the given query, a simplified version of the constraints, to generate the dataset that produces a non-empty result would be as follows (assuming none of the columns are nullable).

```
%Data definition
DATATYPE course_id = CS-101 | BIO-301 | CS-312 | PHY-101 END;
credits:TYPE = SUBTYPE (LAMBDA (x: INT): x > 1 AND x < 11);
course_tuple_type:TYPE = [course_id,credits];
course: ARRAY INT OF course_tuple_type;

%Primary key constraints
ASSERT FORALL(i:course_index, j:course_index):
       course[i].0 = course[j].[0] => course[i].1 = course[j].1

%Foreign key constraints
ASSERT FORALL(i: takes_index):
       EXISTS (j: course_index): takes[i].1 = course[j].0;

%Query conditions
ASSERT course[1].0 = takes[1].1;
ASSERT course[1].1 >= 6;
```

To support nullable columns, we add additional values (outside the domain of the column) for the datatype that correspond to NULL values and explicitly mark those as NULL values. Also, in practice, we found that unfolding the constraints (i.e., specifying the constraints for each tuple instead of FORALL, NOT EXISTS) gives us much better performance [10]. We decide the number of tuples upfront and assert the constraint on each tuple.

2.3 Evaluating Correctness of Student Queries

The dataset generation for each correct query is done once across all students. Based on the datasets generated, XData compares the results of each student query and each correct query provided by the instructor. If all results of the student query are found to match that of a correct query, the student query passes that correct query.

When an instructor specifies multiple correct queries, XData allows the instructor to specify one of the two options

- The additional queries were added to provide more coverage and better testing and all correct queries are equivalent. The student query will need to pass all correct queries for it to be marked as correct.
- The question test provided by the instructor was ambiguous and there could be multiple interpretations of the correct result. In this case, the student query will be marked correct if it passes any one of the correct queries.

The XData system ensures that student queries are safely executed on a different database using temporary tables to ensure that their queries do not interfere with the main database or that the queries of one evaluation do not interfere with another.

3 Edit Based Suggestions For Learning

Generating test data using correct queries provided by the instructor works great for finding student queries with errors. However, once the query is found to be incorrect, the student should be awarded partial marks based on the extent of correctness. It is also useful to make suggestions to the students so that can understand the errors in their queries.

One way to award marks for correctness could have been to use the fraction of datasets (generated by XData) that the student query could pass. This approach turns out to be unfair and could penalize small errors heavily while providing a better score to queries that have more errors. Such examples are shown in [3]. Another way to grade student queries would be to just check for the differences between the correct query and the student query. However, this approach may deduct more marks than required as explained below.

3.1 Approach

The approach we use instead in XData is to compare the student query to each correct query and make changes or edits to the student query to attempt to make it equivalent to the correct query. After each edit, the student query is compared to the correct query and the changes are stopped once the student query is equivalent to the correct query. Checking for equivalence after each edit could have been done using test data generation but that would be very expensive and grading each student query could take minutes. Other approaches for checking equivalence such as Cosette [11] as well as techniques based on tableau [12] work on a limited subset of query constructs and were hence not considered.

Marks are deducted based on the required edits. The edits required are also used to suggest what changes the students should have made to their query to make it correct, thereby providing individualized feedback to each student without any additional human involvement. For each type of query construct being edited, the instructor of the course can specify the weight for that edit. For example, for a query where the instructor is evaluating the student's ability to write aggregations correctly, the instructor may want to deduct more marks for errors in aggregation than for other errors. By default, XData assigns equal weight to each edit being considered.

In general, more than one edit may be needed to make the student query equivalent to the correct query. It is important to note that the order in which the edits are made is also important and that is not sufficient to just compare the query trees of the correct query and the student query to find the changes. One edit to a student query

may allow us to rewrite other parts of the query in a different way. As an example consider the following pair of queries from Section 1.

- Correct query: `SELECT * FROM r INNER JOIN s ON (r.A=s.A) WHERE r.A>10`
- Student query: `SELECT * FROM r INNER JOIN s ON (r.A=s.B) WHERE s.A>10`

There are two differences between the student query and the correct query - the selection condition and the join condition. If the student query is graded just based on these differences, marks corresponding to two edits would be deducted. Even if we grade based on the edits required, if the selection condition is edited first, followed by the join condition 2 edits would be required. On the other hand, if the join condition is edited first and the join condition in the student query is changed to `r.A=s.A`, the selection condition in the student query becomes equivalent to that of the correct query.

3.2 Query Canonicalization

The student and correct query may be written in different ways. For example, the correct query may use a selection condition `A>5` while the student query may write the condition as `NOT(A<=5)`. In order to compare the query structure of the student query to the correct query, we need to make them comparable. The student query and the correct query are made comparable by using canonicalizations.

XData considers two types of canonicalizations:

- **Syntactic Canonicalization**: This is the pre-processing step to reduce irrelevant syntactic differences. These include attribute disambiguation, replacing `NOT, BETWEEN, and WITH` constructs and removing `ORDER BY` from subqueries.
- **Semantic Canonicalization**: In this step, based on the query conditions and the database constraints, the queries are canonicalized semantically. Such canonicalizations include but are not limited to removing distinct clauses based on primary key information and converting outer joins to inner joins based on non-nullable foreign key information.

A detailed list of the canonicalizations is provided in [3]. Such canonicalization rules are often used in query optimizers. However, the goal of the canonicalizations in XData is to get more standard forms of the query.

XData also flattens the query tree where possible (`INNER JOIN, UNION(ALL), INTERSECT(ALL)` as well as predicates involving AND or OR). For the parsed query tree shown in Figure 2, XData would flatten the tree to the one shown in Figure 3. The flattened tree children are compared in an ordered way for non-commutative operators such as `LEFT OUTER JOIN, EXCEPT(ALL)` and `ORDER BY` attribute lists while for commutative operators the order of operands is ignored while matching.

3.3 Edit Sequence Based Grading

XData considers the following form of edits to the flattened tree generated from the student query.

- inserting a node/subtree into the flattened tree
- removing a node/subtree from the flattened tree
- replacing an existing node/subtree from a flattened tree with another node/subtree in the flattened tree
- moving a node/subtree from one position of the flattened tree to another

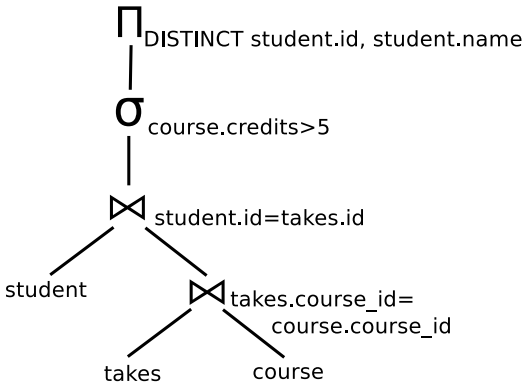


Figure 2: Parsed Tree From Query

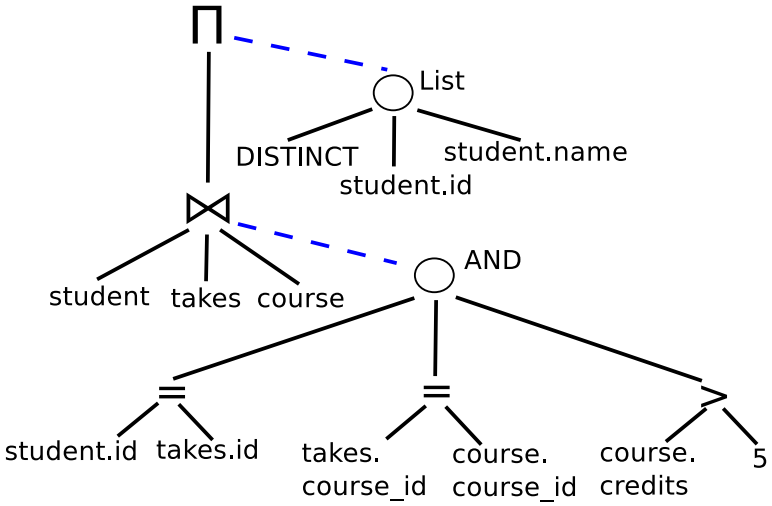


Figure 3: Flattened Tree

When editing a flattened tree generated from a student query, an infinite number of possible edits could be made. However only edits that make the query more similar to the correct query would be useful. In order to add edits that make the student query more similar to the correct query, XData uses the correct query to guide the edits that are generated. The guided edits are based on the differences the student flattened student query tree has with the flattened tree of the correct query. For example, query attributes/conditions/constructs not used in the correct query but present in the student query will be removed when generating edits. For each query edit that XData generates, marks corresponding to the edit as configured by the instructor are deducted. The marks deducted for the edit can be considered the edit cost.

XData can generate multiple guided edits on the student query at each step. From each of these edited queries, more edits are possible. Consider a graph whose nodes are all queries for the given schema. For a student query SQ, edits of the query are also nodes in the graph. Let these edited queries be connected to query SQ with an edge whose weight is the edit cost of the edited query. Canonically equivalent queries, i.e. their canonical forms are the same, are connected by 0 cost edges. The sequence of edits that has the least cumulative cost can now be determined based on the shortest path in this graph from the student query node in the graph to a correct query node. Partial marks can now be awarded based on this shortest path. Since the weight of each edge, which represents the cost of edit is non-negative, the shortest possible path may be found using Dijkstra's shortest path algorithm. Hence, given a set of edits and using a given set of canonicalizations, the shortest path in the graph, as defined above, gives the edit sequence with the least cost. We note that the graph discussed above is for the ease of understanding only and XData does not proactively try to generate the entire graph. In practice, XData generates the nodes of the graph on the fly as needed.

In case the instructor specifies multiple correct queries, the edit sequence based algorithm run based on all correct queries and the best partial marks obtained is awarded.

3.4 Heuristic solution

Even with using only guided edits, the search space is still very large for larger queries if we consider all guided edits to get the shortest path from the student query to the correct query. Hence in practice, we use a greedy heuristic. The heuristic uses a cost benefit model. For each edited query we can get an estimate of how incorrect the query is by finding the differences between the canonicalized versions of the edited query and the correct query. A weighted sum (based on the weight assigned by the instructor for each edit) can be used to find an edit distance which we call the *canonicalized edit distance*. Each guided edit reduces the canonicalized edit distance

to the correct query. The reduction in the canonicalized edit distance from the edit is the benefit of the edit.

For the heuristic algorithm, at each edit step, we find the $benefit - cost$ for each edit. We then pick the edit that has the highest value of the $benefit - cost$ and use it to generate further edits. The remaining edits are discarded at each step. Using the heuristic allows XData to search a much smaller search space. For the incorrect student queries that we had in our course, we found that the heuristic solution works as well and takes orders of magnitude less time as compared to the exhaustive solution [3].

4 Automated Grading Experience

We have successfully used the XData automated grading system across several offerings of undergraduate database courses at IIT Bombay. Before using XData in a course, we empirically confirmed, using results from a previous database course, that XData was able to catch as many as or more errors than when the grading was done manually or when fixed datasets are used for grading. This result was consistent across all questions that XData was able to grade. We found, in several cases, that manual grading had missed subtle errors such as a missing distinct clause.

For the initial course offerings, we used only generated dataset based grading to check for correctness and had to award partial marks manually. The dataset-based grading significantly reduced the human effort involved and allowed us to catch more errors than would have been possible with manual grading. In several cases, however, students were not satisfied since they could not intuitively understand how marks had been deducted or what the error in their specific query was. The tagged datasets on which the student queries failed were shown but often there would be too many of them to understand the specific error. A dataset designed to catch one type of mutation may catch other types of mutations as well and hence it was not always clear what the actual error was. Several students contested the scores that they had been awarded when their query was found to be incorrect. Awarding partial marks manually by the graders was still tedious since students often wrote queries in very different and complex ways. Manually transforming such student queries to a simpler form was difficult. For instance, in one case a correct query involved using a NOT IN clause and some students used a combination of multiple EXCEPT and INTERSECT clauses.

When using partial marking in combination with dataset based evaluation, we reduced the human effort as well as provided much better feedback. We experienced fewer students contest grades when it was assigned automatically compared to when a human would manually assign grades. The edit-based guidance was also useful for students to understand where they went wrong. For the first course setting where we used automated partial marking for evaluation, we found that across 1800 student query submissions that were graded by XData, only 2 queries were contested by students. In both cases, we traced back the errors in grade to bugs in our code. One of the main challenges when using automated grading was to provide sufficient types of correct queries that covered the student queries.

Since the edit sequence based guided edits provide a way to change an incorrect student query to a correct query, the guided edits can be used by students to learn the mistakes that they made and how the mistakes could have been corrected. Such feedback was very helpful especially for beginners to understand how to write correct SQL queries.

5 Related Work

There has been a significant interest in testing query equivalence or correctness and in automated grading. Related work include the following.

Checking Query Equivalence

XData uses test data generation based on mutation testing to generate test datasets to check the equivalence of the correct query and student query. Tuya et al. [13] describe a number of possible mutations for SQL queries. However, they do not handle test data generation for killing these mutations. Other approaches on testing query equivalence using datasets include Qex [14] and SQL full predicate coverage by Riva et al. [15]. Test data generation in these systems is aimed at testing SQL queries in database applications and they consider only a limited subset of SQL query constructs.

Techniques based on tableau [16] and its extensions [12, 17] can be used to check for query equivalence for a restricted class of conjunctive queries. Cosette [11] and U-semiring [18] can also be used to check for SQL semantic equivalence using a restricted set of axioms. SPES [19] uses a symbolic approach for checking query equivalence on SQL queries under bag semantics for select-project-join(SPJ) queries as well as aggregate and union queries.

Grading SQL Queries

The Gradiance system [20] provides multiple choice type questions where the instructor has to provide some correct as well as incorrect answers and explanations of incorrectness. In such assignment settings, since the students are only able to select from limited options, subtle mistakes that students could make may not always be covered by the incorrect answers. Gradescope [21], uses a fixed dataset to evaluate the correctness of student SQL queries. As discussed earlier, using fixed datasets may miss errors and would not be able to provide any meaningful feedback to incorrect student queries. RATest [22] provides feedback for incorrect queries by deriving small datasets that produce different results in a student query as compared to a correct query. I-Rex [23] allows users to trace the SQL query evaluation for each constituent block in the query execution.

Automated Grading for Programming Assignments

Grading programming assignments has some similar challenges as grading database queries. CPSGrader [24] can grade programming assignments for cyber-physical systems using constraints synthesis and uses reference solutions to provide feedback. AutoGrader [25] can grade introductory level python programs and provide student feedback using program synthesis and high-level error modeling specifications. However, AutoGrader can only model specific predictable errors.

SARFGEN [26] provides feedback to student queries by aligning student programs to similar correct reference programs and finds the minimum number of edits to the student program to match the chosen reference program. This approach is similar to our approach of edit-based grading. However, SQL queries have database constraints that are not part of the query but need to be accounted for during edits and equivalence checking. Hence we have a more complex semantic canonicalization step that can take into account constraints such as primary keys and foreign keys.

6 Conclusion and Open Challenges

The XData system is very useful for grading assignments on SQL queries and for providing feedback to students about mistakes they made. For large classes or online courses, such automated grading systems are essential and the automated individualized feedback would be very useful to new learners. Our experience in using the grading system has been very positive from both the teaching assistants as well as the students. The XData grading system as well as the source code are available for download from `http://www.cse.iitb.ac.in/infolab/xdata`.

One of the key challenges in the XData grading scheme is the number of ways a correct SQL query can be written. We have found that students sometimes write queries using a very different approach than we had

anticipated. Catching errors with such approaches as well as awarding partial marks in these cases may be challenging. One way to address these would be to automatically group student query submissions and generate datasets for one student query in each group. We could then use the datasets to identify correct queries from the group and automatically add these as additional correct queries for grading. Another key challenge is in dealing with student queries that have additional query constructs that do not affect the query results. One of the cases that we found in our course was when a student had used the query `Q UNION Q'` where Q' was always empty. The query had one error in Q but marks were also deducted based on edit for Q'.

Acknowledgments: We thank all students and researchers who worked on the XData project as well as those who used the system and provided their valuable feedback. This work was partially supported by research funding and a PhD fellowship from Tata Consultancy Services.

References

[1] B. Chandra, B. Chawda, B. Kar, K. V. M. Reddy, S. Shah, and S. Sudarshan, "Data generation for testing and grading SQL queries," *VLDB Journal*, vol. 24, no. 6, pp. 731–755, 2015.

[2] A. Bhangadiya, B. Chandra, B. Kar, B. Radhakrishnan, K. V. M. Reddy, S. Shah, and S. Sudarshan, "The XDa-TA system for automated grading of SQL query assignments," in *International Conference on Data Engineering (ICDE)*, 2015.

[3] B. Chandra, A. Banerjee, U. Hazra, M. Joseph, and S. Sudarshan, "Edit based grading of SQL queries," in *CODS-COMAD 2021: 8th ACM IKDD CODS and 26th COMAD*, 2021, pp. 56–64.

[4] P. Agrawal, B. Chandra, K. V. Emani, N. Garg, and S. Sudarshan, "Test data generation for database applications," in *International Conference on Data Engineering (ICDE)*, 2018, pp. 1621–1624.

[5] A. J. Offutt, "A practical system for mutation testing: Help for the common programmer," in *International Conference on Test (ICT)*, 1994, pp. 824–830.

[6] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. IOS Press, 2009, vol. 4, ch. 8.

[7] C. Barrett and C. Tinelli, "CVC3," in *Computer Aided Verification (CAV)*, 2007, pp. 298–302.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 171–177.

[9] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[10] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. P. Gupta, and D. Vira, "Generating test data for killing SQL mutants: A constraint-based approach," in *International Conference on Data Engineering (ICDE)*, 2011.

[11] S. Chu, C. Wang, K. Weitz, and A. Cheung, "Cosette: An Automated Prover for SQL," in *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[12] Y. E. Ioannidis and R. Ramakrishnan, "Containment of conjunctive queries: Beyond relations as sets," in *ACM Transactions on Database Systems (TODS)*, vol. 20, no. 3, 1995, pp. 288–324.

[13] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva, "Mutating database queries," in *Information and Software Technology*, 2007, pp. 398–417.

[14] M. Veanes, N. Tillmann, and J. de Halleux, "Qex: Symbolic SQL Query Explorer," in *Logic Programming and Automated Reasoning (LPAR)*, 2010, pp. 425–446.

[15] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya, "Constraint-based test database generation for SQL queries," in *Workshop on Automation of Software Test*, ser. AST '10, 2010, pp. 67–74.

[16] A. V. Aho, Y. Sagiv, and J. D. Ullman, "Equivalences among relational expressions," *SIAM Journal on Computing (SICOMP)*, vol. 8, no. 2, pp. 218–246, 1979.

[17] Y. Sagiv and M. Yannakakis, "Equivalence among relational expressions with the union and difference operation," in *International Conference on Very Large Data Bases (VLDB)*, 1978, pp. 535–548.

[18] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu, "Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries," in *Proceedings of the VLDB Endowment (PVLDB)*, vol. 11, no. 11, 2018, pp. 1482–1495.

[19] Q. Zhou, J. Arulraj, S. Navathe, W. Harris, and J. Wu, "SPES: A symbolic approach to proving query equivalence under bag semantics," in *International Conference on Data Engineering (ICDE)*, 2022.

[20] "Gradiance: The Gradiance service for database systems," http://www.gradiance.com/db.html (Retrieved on Aug. 1, 2022).

[21] "Gradescope," https://www.gradescope.com (Retrieved on Aug. 1, 2022).

[22] Z. Miao, S. Roy, and J. Yang, "Explaining wrong queries using small examples," in *ACM SIGMOD International Conference on Management of Data (SIGMOD)"*, 2019, pp. 503–520.

[23] Y. Hu, Z. Miao, Z. Leong, H. Lim, Z. Zheng, S. Roy, K. Stephens-Martinez, and J. Yang, "I-rex: An interactive relational query debugger for SQL," in *SIGCSE 2022: The 53rd ACM Technical Symposium on Computer Science Education*, 2022, p. 1180.

[24] G. Juniwal, A. Donzé, J. C. Jensen, and S. A. Seshia, "CPSGrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory," in *International Conference on Embedded Software (EMSOFT)*, 2014.

[25] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013, pp. 15–26.

[26] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 481–495.

Towards Technology-Enabled Learning of Relational Query Processing

Sourav S. Bhowmick ⋆ and Hui Li ◇
⋆ Nanyang Technological University (NTU), Singapore, assourav@ntu.edu.sg
◇ Xidian University, China, hli@xidian.edu.cn

Abstract

*The database systems course has gained increasing prominence in academic institutions due to the convergence of widespread usage of relational database management system (*RDBMS*) in the commercial world, the growth of Data Science, and the increasing importance of lifelong learning. A key learning goal of learners taking such a course is to learn how* SQL *queries are processed in an* RDBMS *in* practice*. Most database courses supplement traditional modes of teaching with technologies such as off-the-shelf* RDBMS *to provide hands-on opportunities to learn database concepts used in practice. Unfortunately, these systems are not designed for effective and efficient pedagogical support for the topic of relational query processing. In this vision paper, we identify novel problems and challenges that need to be addressed in order to provide effective and efficient technological supports for learning this topic. We also identify opportunities for data-driven education brought by any effective solutions to these problems. Lastly, we briefly report the* TRUSS *system that we are currently building to address these challenges.*

1 Introduction

Learning is the acquisition of knowledge or skills through study, experience, or being taught [19]. It is not just listening and accepting what we are taught, but understanding and experiencing them. Education, on the other hand, is the acquisition of knowledge through a process of receiving or giving systematic instruction. Hence, although learning and education are closely related, the former has a broader scope and impact. Specifically, learning can be facilitated through education, personal development, schooling, training or experience. It is not limited to a certain age or period in life. Indeed, while formal education for young adult learners at universities has been the focus of educational provisions in the industrial age, the digital age is now seeing an increased experimentation of "lifelong learning" [6] with provisions such as work-study programmes for early career and mid-career individuals, and digital learning initiatives.

The growing demand for lifelong learning coupled with the widespread use of relational database management system (RDBMS) in the commercial world and the growth of Data Science as a discipline have generated increasing demand of database-related courses in academic institutions. Learners from diverse fields and experiences aspire to take these courses, even with limited Computer Science backgrounds [26]. In a computer science degree program, the key goal of a database systems course is to teach learners how to *build* a database system. On the other hand, the focus of the course in a data science program is to be able to *control* a database system effectively.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

To facilitate both these goals, it is paramount for learners to learn how SQL queries are processed in an RDBMS in practice. Traditionally, this learning goal is achieved through textbooks and lectures. Specifically, major database textbooks [17, 36] introduce *general* (*i.e.,* not tied to any specific RDBMS) theories and principles associated with relational query processing and optimization using natural language-based narratives and visual examples. This allows a learner to gain a general understanding of SQL query execution strategies.

It is well-established in education that *effective* use of technology has a positive impact on learning [24]. It causes learners to be more motivated and engaged, thus, enabling them to retain more information. It also increases hands-on learning opportunities. In fact, technology is best used as *"a supplement to normal teaching rather than as a replacement for it"* [24]. Hence, in order to promote effective and efficient learning for diverse individuals in full recognition of the complexity of the topic of relational query processing, *learner-friendly* tools are paramount to augment the traditional modes of learning (*i.e.,* textbook, lecture). Indeed, database systems courses in major institutions around the world supplement traditional style of learning with the usage of off-the-shelf RDBMS. Unfortunately, these RDBMS are not designed for pedagogical support. Although they enable hands-on learning opportunities to build database applications and pose a wide variety of SQL queries over it, very limited effective and efficient learner-friendly support, beyond the visualization of *query execution plans*, is provided for understanding and experiencing the processing and optimization of these queries *in practice* by the underlying relational query engine.

Given the challenges faced by learners to learn SQL [34], there has been increasing research efforts to build tools and techniques to facilitate comprehension of complex SQL statements [15, 25, 29, 30, 32, 33], automated grading of SQL queries [8], and so on. However, scant attention has been paid to explore technologies that can enable learning of relational query processing [21, 31, 42]. In this paper, we articulate a vision shaped by the following fundamental questions: (a) What are the key problems that we need to address to facilitate technology-enabled learning of relational query processing in practice? (b) How can the potential solutions to these problems support data-driven education with the goal of making teaching and learning practices more effective and efficient? Specifically, our vision calls for a *learning-centric*, *generic*, and *psychology-aware* approach grounded on the *theories* of motivation and learning to address these challenges. Note that by no means we claim that the list of problems discussed in this article is exhaustive. The pervasive desire here is to galvanize the data management community to explore this nascent inter-disciplinary topic at the intersection of learning sciences and data management by leveraging these problems as the seed.

The rest of the paper is organized as follows. In Section 2, we briefly introduce relevant *motivation theories* that are at the foundation of learning and may potentially impact the design of any learner-centric, technology-enabled solutions for learning. Section 3 introduces the key novel research challenges that need to be addressed in order to realize the vision. We briefly report the TRUSS system which we are currently building to address these challenges in Section 4. The last section concludes this paper.

2 Motivation Theories

Learning is impacted by *motivation*, which is the process that initiates, guides, and maintains goal-oriented behavior [27]. Specifically, motivation impacts how likely a learner is willing to learn. Since effective use of technology in learning has positive impact on motivation, we need to understand motivation theories and their impact on learning. These theories should underpin any effective technological solutions for facilitating learning. In this section, we first briefly describe key theories related to motivation proposed in the domain of education psychology. Next, we highlight how these theories can guide technology-enabled solutions for learning. In the next section, we shall identify novel research issues that are grounded on these theories to facilitate learning of relational query processing.

Motivation theories. Motivation can be broadly characterised into two types, *intrinsic* and *extrinsic* [37]. *Intrinsic motivation* is the act of doing an activity purely for the joy of doing it whereas *extrinsic motivation* is to do

something due to the influence of external rewards or punishments (*e.g.,* good grades, jobs). Although the latter is not optimal for learning, research suggests that extrinsic motivation is prevalent [16, 37] and may pave the way to intrinsic motivation. That is, a learner may initiate learning due to external factors but the motivation may morph to an intrinsic one during task engagement.

Regardless of the type of motivation, *Achievement Goal Theory* [7] argues that all motivation are essentially linked to one's goals that can take two forms, namely *performance goals* and *mastery goals*. The former is caused by satisfaction of one's ego (*e.g.,* appearing superior to one's peers) whereas the latter is aligned with intrinsic motivation and is grounded on the pure desire to master a skill or concept. The *Expectancy Value Theory* [43] postulates that expectation and value of learning a skill or concept have direct impact on the performance and task choice of a learner. To elaborate further, an individual's effort and performance for a task are influenced by their expectation of success or failure. Note that expectations and values themselves are influenced by how a learner assess their competency and perceived task difficulty. If a learner has felt satisfaction in undertaking a similar task in the past, then it is more likely they will put effort to the current task. However, if past experience shows the task is either too difficult to be completed or not sufficiently difficult enough then they may not engage with it. The *Flow Theory* [35] describes a psychological state in which an individual is purely intrinsically motivated to learn without any external factors. Such state is said to occur when the task is neither too difficult to cause helplessness or frustration nor too easy to make a learner bored.

Motivation theories in technology-enabled learning frameworks. Every learner has intrinsic or extrinsic motivation to reach their learning goals. Hence, any technology-enabled learning framework should be cognizant of the above theories in order to cultivate motivation to learn. Specifically, technologies to supplement learning of relational query processing and optimization should support the followings:

- Learners may learn relational query processing due to intrinsic or extrinsic motivation. It is important that any technological framework support both and facilitate transformation of extrinsic motivation to intrinsic one during the course of interacting with it.
- Increase learners' satisfaction of successful completion of the task of learning relational query processing as well as facilitate flow state of learners to move to the Goldilock's zone (*i.e.,* learning relational query processing concepts is neither too difficult nor too easy).

3 Research Issues

In this section, we outline novel *learner-centric* research issues in the burgeoning topic of technology-enabled learning of relational query processing. It is worth noting that although technology engages and motivates learners, it is advantageous for learning only when it is aligned to with what is to be learned [24]. Hence, the issues we focus for technological support are learning about query execution plans, exploration of *alternative query plans* in a plan space, and cost estimation of a physical query plan. Observe that all these issues are typically covered by a database systems course. A common theme that cuts across these issues is the pervasive desire for motivation theory-aware tools and techniques that aim to *supplement* existing off-the-shelf RDBMS to facilitate learning of these issues. We begin with a brief background on relational query plans that learners typically encounter in a database systems course.

3.1 Query Plans

Given an arbitrary SQL query, an RDBMS generates a *query execution plan* (QEP) to execute it. A QEP consists of a collection of physical operators organized in form of a tree, namely the *physical operator tree* (*operator tree* for brevity). Figure 1(a) depicts an example QEP with a collection of physical operators. Each physical operator, *e.g.,* `SEQUENTIAL SCAN`, `INDEX SCAN`, takes as input one or more data streams and produces an output one. A QEP

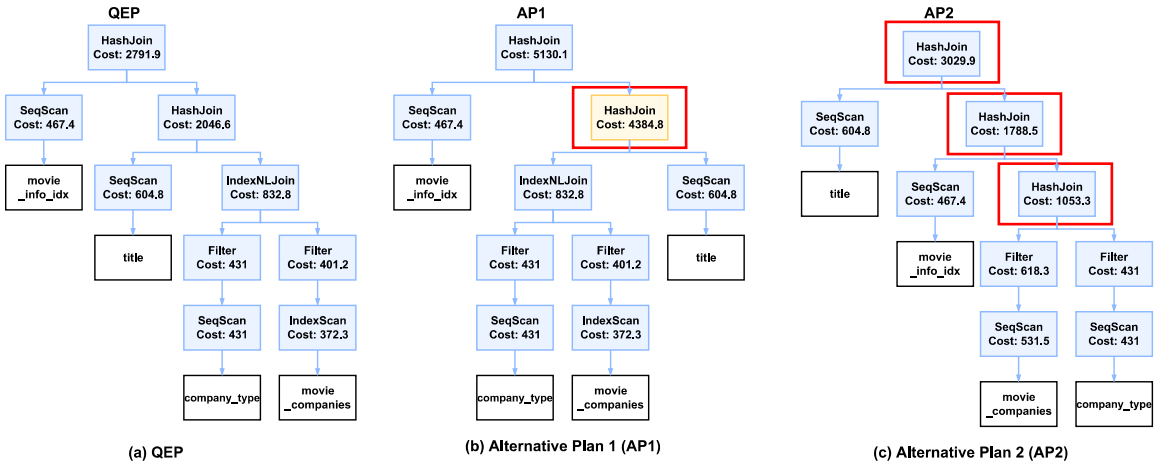


Figure 1: Examples of QEP and alternative query plans.

explicitly describes how the underlying RDBMS shall execute the given query. Notably, given an SQL query, there are many different query plans, other than the QEP, for executing it. We refer to these different plans (other than the QEP) as *alternative query plans* (AQP). Figures 1(b)-(c) depict two examples of AQP.

3.2 Understanding Query Execution Plans (QEPs)

A key goal of learning relational query processing is to understand how SQL queries are processed in an RDBMS in practice. This can be achieved by understanding the content of the QEP of a given query. Major database textbooks (*e.g.,* [17, 36]) typically illustrate QEPs of simple SQL queries, their costs, and adverse impact on estimated cost if alternative physical operators are chosen (*e.g.,* merge join instead of hash join) for a QEP. However, they are not interactive and the variety of examples they can discuss is constraint by the page limit and cost. Hence, only a very few simple, static examples of QEPs are typically exposed to learners.

Off-the-shelf RDBMS (*e.g.,* PostgreSQL) provide the opportunity to greatly mitigate the limitations of textbooks. A learner may implement a database application in an RDBMS, pose queries over it, and peruse the associated QEPs to comprehend how they are processed by an industrial-strength query engine in practice. Such interactivity provides learners the freedom to formulate a large number of queries with diverse complexities and view the contents of corresponding QEPs in real-time.

Most existing RDBMS expose the QEP of an SQL query using *visual* or *textual* (*e.g.,* unstructured text, JSON, XML) format. Unfortunately, comprehending these semistructured textual formats to learn about query execution strategies of SQL queries in practice can be daunting for learners. In contrast to natural language-based narrations in database textbooks, they are not user-friendly and assume deep knowledge of vendor-specific implementation details. On the other hand, the visual format is relatively more user-friendly but hides important details. Consequently, in consistent with the *Expectancy-Value Theory*, the task difficulty may become a barrier for some learners to learn about query execution strategies in a specific RDBMS from these QEP formats.

Example 1: Doreen is an undergraduate student in a data science program who is currently enrolled in a database course. She wishes to understand the execution steps of the following SQL query in PostgreSQL on the IMDb benchmark dataset [1] by perusing the corresponding QEP in Figure 2(a) (partial view).

```
SELECT mc.note AS production_note,
   t.title AS movie_title,
   t.production_year AS movie_year
FROM company_type AS ct,
```

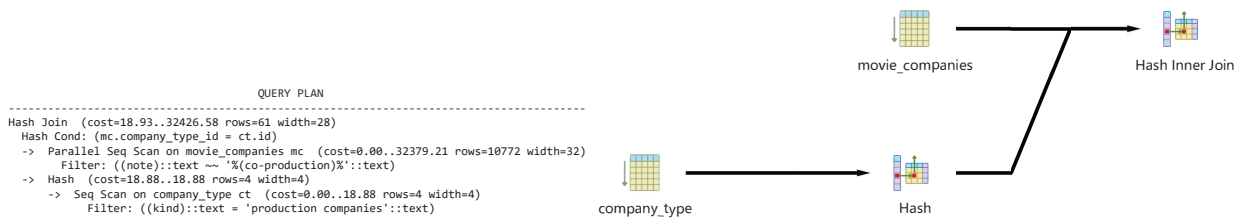


Figure 2: A QEP in PostgreSQL and its visual tree representation.

```
    movie_companies AS mc,
    movie_info_idx AS mi_idx,
    title AS t
  WHERE ct.kind = 'production companies'
    and mc.note like '%(co-production)%'
    AND ct.id = mc.company_type_id
    AND t.id = mc.movie_id
        AND mc.movie_id = mi_idx.movie_id;
```

Unfortunately, Doreen finds it difficult to mentally construct a narrative of the overall execution steps by simply perusing it. This problem is further aggravated in more complex SQL queries. Hence, she switches to the visual tree representation of the QEP as shown in Figure 2(b). Although relatively succinct, it simply depicts the sequence of operators used for processing the query, hiding additional details about the query execution (*e.g.,* sequential scan, join conditions). In fact, Doreen needs to manually delve into details associated with each node in the tree for further information. ∎

Since natural language (NL)-based narratives aided with visual examples (as in textbooks and lectures) have been the traditional mode of learning for decades, we advocate that an intuitive natural language (NL)-based description of a QEP can greatly augment learning of the execution strategies of SQL queries by an RDBMS. The intuition is that NL-based descriptions may facilitate the flow state of learners (*Flow Theory*) to the Goldilock's zone. This can then effectively complement the current visual tree format generated by existing RDBMS. Specifically, a learner may either use the visual QEP to get a quick overview and then peruse the NL description or study them in parallel to acquire detailed understanding. To support this hypothesis, we surveyed 62 and 56 unpaid volunteers taking the undergraduate database course in NTU in two semesters (2019 and 2020). We use the TPC-H v2.17.3 benchmark and a rule-based natural language generation tool for QEPs [31] to generate natural language descriptions of QEPs for SQL queries formulated by the volunteers. The volunteers were asked: *"which query plan format they prefer for learning?"* 53.2% and 55.4% preferred NL-based description, respectively. Very few (3% and 5.4%, respectively) preferred the text format. Hence, there is clear evidence that learners prefer to use NL-based and visual tree-based formats for learning about QEP.

The majority of NL interfaces for RDBMS [28], however, have focused either on translating natural language sentences to SQL queries or narrating SQL queries in a natural language. Scant attention has been paid for generating natural language descriptions of QEPs [31, 42], which is a challenging problem. Although deep learning techniques, which can learn task-specific representation of input data, are particularly effective for natural language processing, it has a major upfront cost. These techniques need massive training sets of labeled examples to learn from. Such training sets in our context are prohibitively expensive to create as they demand database experts to translate thousands of QEPs of a wide variety of SQL queries. Even labeling using crowdsourcing is challenging as accurate natural language descriptions demand experts who understand QEPs. Note that accuracy is critical here as low quality translation may adversely impact individuals' learning.

3.3 Learning Impact of Alternative Choices on QEP

Natural language description of a QEP enables a learner to understand the execution steps of a query. This may pique the interest of a learner to raise further questions related to query processing centred around a specific QEP. Since major database textbooks typically discuss the adverse impact of choosing alternative physical operators or join ordering on the estimated cost, a learner may also like to delve deeper into the impact of these alternative choices on the estimated query processing cost of their queries.

Example 2: Reconsider Example 1. Doreen's course lectures and textbook discuss the impact of physical operator choices and join ordering on the selection of a QEP. Hence, after perusing the content of the QEP, she wonders what will be the impact on the cost be if the hash join is replaced by a merge join? Is the estimated cost of the alternative query plan substantially higher compared to the QEP? How much is the impact on the estimated cost if the join ordering is changed? In this context, a narrative that explains why the QEP is chosen by connecting its content with knowledge garnered from database textbooks will greatly benefit her learning. ∎

Unfortunately, as stated earlier, off-the-shelf RDBMS are not developed for pedagogical support. Typically, they do not expose the impact of alternative choices of various physical operators or join ordering on the QEP in a *user-friendly* manner to aid learning. Note that such information is invaluable to learners as it not only facilitates hands-on inquire-driven learning on the impact of a choice of a physical operator or a specific join ordering on the estimated cost of a QEP but it also enables them to comprehend why a QEP is chosen by the underlying RDBMS. However, an RDBMS typically demands a learner to manually pose SQL queries with various constraints on *configuration parameters* (*e.g.*, `enable_hashjoin`, `enable_nestloop` in PostgreSQL) to view the corresponding QEP containing specific physical operators. Furthermore, one has to manually compare the generated plan with the original QEP to understand the impact. Notably, a database course may not introduce these configuration parameters while exposing syntax and semantics of SQL. It is also impractical to assume that learners will be familiar with them when many are taking the course for the first time. Clearly, based on the *Expectancy-Value* and *Flow* theories, a learner-friendly framework that can facilitate exploration of the impact of various physical operators and join ordering on a QEP can greatly motivate learners to deeper engagement and learning of this topic.

Intuitively, given an SQL query and learner-specified *preferences* (*e.g.,* merge join, index scan, specific join ordering), the goal is to automatically visualize the impact of these choices on the selected QEP. In this context, it is important to generate a natural language-based explanation that goes beyond the conventional least-cost-based explanation to connect established knowledge related to usage scenarios of different physical operators from textbooks with the specified preferences. For instance, examples of some established knowledge are: (a) index scan is the optimal access path for low selectivity whereas sequential scans perform better in high selectivity [11]; (b) merge join is preferred if the join inputs are large and are sorted on their join column [2]; (c) nested-loop join is ideal when one join input is small (*e.g.,* fewer than 10 rows) and the other join input is large and indexed on its join columns [2]. Such knowledge in the form of explanations will naturally facilitate learners' understanding of relational query processing. For instance, consider Example 2. Explanations such as (b) will help Doreen to understand why a hash join was chosen by the relational query engine.

The problem is challenging from several fronts. While under-the-hood it is straightforward to generate an SQL query involving the learner-specified preferences and retrieve the corresponding QEP, automatically generating appropriate visualization framework to aid learning is challenging. First, how should the results be presented in consistent with motivation theories to motivate learners to explore and learn? Note that a learner may want to view the impact of multiple physical operators and join ordering together instead of just a single operator or join ordering. Simply generating an NL description of the AQP is insufficient since this will demand a learner to manually compare the description of the original QEP with it in order to understand the impact of various operators and join ordering. Naturally, this becomes tedious especially for complex queries. Second, how can we generate NL explanations that augment learning by connecting with textbook knowledge? It demands

sophisticated text extraction, analytics, and summarization framework that connects the alternative query plans with relevant established knowledge embedded in online resources.

3.4 Exploration of Informative Alternative Query Plans

In the preceding challenge, a learner has clear preferences that they wish to explore with respect to a QEP. However, this may not always be the case. Some learners may not have clear idea of what alternative query plans they are interested in.

Example 3: Meng is another undergraduate student pursuing a degree in computer science and a classmate of Doreen in the database course. He also formulates the query in Example 1. After viewing the QEP, he wonders what are the different alternative query plans (AQPs) considered by the underlying RDBMS during the QEP selection process. Specifically, are there alternative plan(s) that have similar (resp. different) structure and physical operators but very different (resp. similar) estimated cost? If there are, then how they look like? ∎

Off-the-shelf RDBMS do not expose a *representative* set of alternative query plans considered by the underlying query optimizer during the selection of a QEP in a *user-friendly* manner to aid learning. Hence, due to the lack of easy access to such information in RDBMS, based on the *Expectancy-Value Theory*, learners may restrict themselves to the simple and limited number of examples that are typically exposed in textbooks and lectures or simply abandon the effort. Clearly, a learner-friendly framework that can facilitate retrieval and exploration of "informative" alternative query plans associated with a given query can greatly aid in answering Meng's questions related to the query optimization process.

Selecting a set of *informative* AQPs to facilitate learning is a technically challenging problem. First, what is an "informative" AQP in the context of learning? To elaborate further, reconsider Example 3. Figures 1(b)-(c) depict two alternative plans for the query where the physical operator/join order differences are highlighted with red rectangles and significant cost differences are shown using yellow nodes. Specifically, *AP1* has very similar structure as the QEP but different join order involving `title` and `company_type` relations and significantly different estimated cost. *AP2*, on the other hand, displays similar estimated cost as the QEP but different join order. *Which of these alternative plans should be revealed to Meng?* The overarching goal here is to choose alternative plan(s) that may enhance Meng's knowledge of the QEP selection process (*i.e.,* informative) as well as motivate him to learn and explore. Certainly, any *informativeness* measure needs to be cognizant of plans that a learner have already viewed for her query (including the QEP) in order to avoid the exposure of highly similar information. It should also facilitate retrieval of plans that learners may be interested in as far as query optimization is concerned. Hence, it is paramount to take feedback from learners on the *types* of AQPs that are potentially of interest to them and devise a mechanism to quantify *informativeness* of a plan by mapping the knowledge acquired from the feedback to a *utility* measure. Subsequently, we need to design techniques that can select informative plans that *maximize* the *utility* as we cannot simply rely only on the estimated cost of alternative plans. Second, the number of candidate AQPs for a given SQL query is exponential in the worst case [13]. Hence, it is prohibitively expensive to scan all these plans to select informative ones. Note that the selection of AQP cannot be integrated into the plan enumeration step of the underlying query optimizer. We need to know the QEP when computing AQP as they are selected with respect to the QEP a learner has seen.

At first glance, it may seem that we can select $k > 1$ alternative query plans where k is a value specified by a learner. Although this is a realistic assumption for many top-k problems, learners may not necessarily be confident to specify the value of k always. They may prefer to *iteratively* view one plan-at-a-time and only cease exploration once they are satisfied with the understanding of the query optimization process for a specific query. Hence, k may not only be unknown *apriori* but also the selection of an AQP at each iteration to enhance learning of different plan choices depends on the plans viewed by a learner thus far. Clearly, it does not increase learners' understanding of the query optimization process or motivate them to use the framework if a plan with highly

similar information of an already viewed plan is revealed to them in the subsequent iterations. This demands for a flexible solution framework that can select informative AQPs in absence or presence of the k value.

3.5 Understanding Cost Estimation of a Physical Query Plan

The preceding subsections introduce research issues that aim to facilitate learning of the execution strategy of an SQL query and interesting plan choices a relational query optimizer makes in practice. Another key knowledge that a learner needs to acquire is the cost estimation procedure of these plans.

Example 4: Doreen and Meng have learnt from textbooks and lectures how the cost of a physical query plan can be estimated. However, the number of queries considered in these modes of learning and their complexities are limited. They are motivated to experience cost estimation of plans associated with a wider variety of queries. Hence, they pose several queries with different degrees of complexity on the IMDb dataset in PostgreSQL. They can view the overall estimated cost of a QEP as well as cost of different subtrees (*e.g.,* Figure 1). However, they cannot view step-by-step details of the input parameters and the formulas used by the underlying query optimizer to compute these numbers. For instance, in Figure 1(a), why is the cost of the first `HASH JOIN` *2046.6*? By undertaking a back-of-the-envelope calculation using formulas learnt in the course, they could not replicate this value. Are some of the principles and formulas to compute cost different from what they have learnt from textbooks and lectures? If so, then why?

Doreen and Meng also wonder what the intermediate result sizes of different operations are here? When they execute one of the queries, they have to wait for a considerable amount of time to view the results. Does the estimated time cost of the QEP differ significantly from the actual cost? Why? ∎

Existing RDBMS do not provide any learner-friendly support to facilitate such learning. Consequently, based on motivation theories, learners may not pursue this direction of inquiry using an RDBMS, surrendering valuable opportunity for hands-on acquisition of knowledge of the cost estimation process. It is challenging, however, to expose an interface to facilitate such learning and exploration. First, it demands automated analysis of the code base of the underlying query optimizer to extract various formulas used for cost estimation. These formulas may not necessarily be identical across all RDBMS or textbooks. For instance, in [17], the cost of a selection involving inequality condition is approximated to be 1/3 of the input size independent of the selection condition. On the other hand, in [36], more accurate measure is used for estimating the selection cost. Furthermore, a specific RDBMS may implement variants of these formulas. Second, it is paramount to connect these formulas with specific input parameters for a query to reveal how the cost of a plan is estimated while emphasizing the similarity and differences with textbook knowledge. A framework that can support this in a palatable manner to facilitate learning is non-trivial as it may demand a sophisticated natural language generation framework that connects analysis of the code base with textbook content. Third, superior visualization and NL-based framework are necessary to explain to learners the reasons for the differences in estimated and actual cost of a query. Although tools such as [21] allow one to visualize the cost of different plans over the plan space, they are not designed for explaining the *cost differences* for a *specific* query in a palatable manner.

3.6 A Unifying Framework: Chatting with a Relational Query Engine

Although addressing the aforementioned issues has the potential to facilitate learning of relational query processing by providing learner-friendly platforms, a set of isolated platforms that address these different issues will make it cumbersome for learners to navigate and take advantage of them. For instance, learners may find it overwhelming to operate three independent technology-enabled learning platforms targeting NL description generation of QEP, exploration of AQP, and cost analysis of query plans, respectively. This may deter them to use such technology for learning. Given that young adults often interact through chat apps (*e.g.,* Whatsapp, WeChat), a natural language interaction framework (*i.e.,* chatbot) that can unify these solutions may bring practical benefits to learners. A

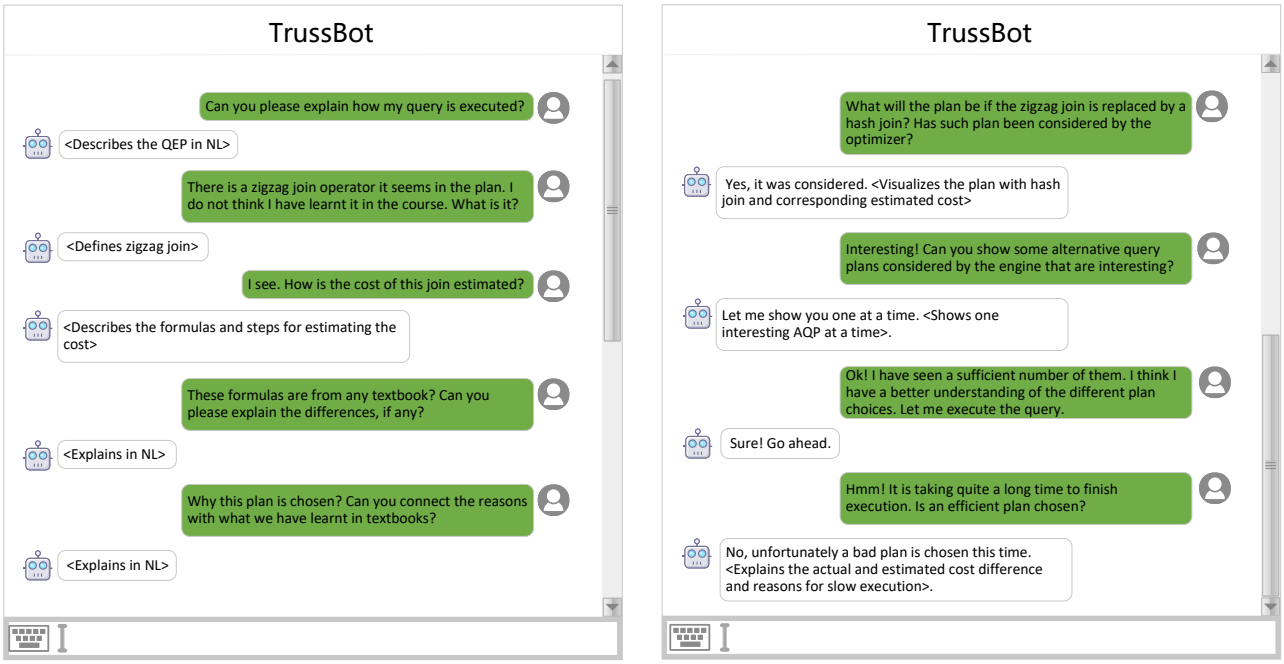


Figure 3: An example of interaction between a learner and the chatbot.

possible interaction between a learner and a hypothetical chatbot designed for interacting with a relational query engine are shown in Figure 3.

Building a high-quality chatbot for a relational query engine to facilitate learning is non-trivial and challenging. In addition to the challenges mentioned earlier for addressing individual components, a chatbot brings new challenges with respect to correct parsing and interpretation of a learner's statement, constructing correct (syntactically and semantically) responses in a natural language, engaging learners in conversations, and so on. While these challenges are long recognized in building a generic chatbot [20], the domain-specific nature of the problem brings in interesting flavor to it. Different from natural conversations, a learner's questions usually have concrete objectives, actively requesting information, and an answer to every question should facilitate understanding of relational query processing. Furthermore, a learner's questions at each step are not only closely related to the chatbot's current answers, but also need to take into account the context of the previous parts of the conversation. For example, consider the first three questions in Figure 3 from the learner. These questions are actively requesting information related to relational query processing. Observe that the third question is related to the preceding parts of the conversation.

Any chatbot needs to consider two kinds of information in a learner's question: (1) the intent of the question (*e.g.,* understanding cost computation) and (2) the content of the question (*e.g.,* cost computation steps of zigzag join in a QEP). To this end, we can construct a *query processing knowledge graph* semi-automatically to represent a collection of relational query processing concepts. Then the intent and content can be determined by *mapping* the question to different concepts in the knowledge graph. Note that similar idea of knowledge graph has been recently exploited in the context of question generation for multi-party court debates for judicial education [44]. Once the intent and content of a question are determined, the chatbot invokes the relevant component (Section 3.2-3.5) to retrieve the answer for the specific question. The result returned by it is then *transformed* into a natural language (supported by visual representations, if necessary) and presented to the learner.

Table 1: Learning-centric issues.

Issue	Questions to address
Rational for the impact of technologies on learning	Will learners work more efficiently, more effectively, more intensely? Will the technology help them to learn for longer, more deeply, more productively?
Role of technology in learning	Will it help learners to gain access to learning content? Will the technology provide feedback?
Technology should support effective interaction for learning	Does it support effective interaction with learners?
Identify what learners will stop doing	What it will replace or how the technology activities will be additional to what learners would normally experience?

3.7 Learning-centric, Generic, and Psychology-Awareness of Solutions

In addition to the challenges within each aforementioned issues, any solution to them must ensure the following features.

- **Learning-centric.** The role, impact, and interaction of the platforms designed to address aforementioned issues have to be learning-centric, *i.e.,* they bring about improvement in learning. Table 1 lists the learning-centric issues [24] that any technology-enabled solution needs to address. For instance, consider the last issue. An effective solution to NL descriptions of QEPs will provide learners an additional interface to learn about query execution strategies to what they would normally experience. Similarly, consider the second issue. A solution to user-friendly exploration of AQPs will enable learners to gain easy access to informative plans to aid learning of the query optimization process.
- **Generalizability.** Solutions must be *generalizable* to different RDBMS and applications. This will significantly reduce the cost of its deployment in different learning institutes and environments where different application-specific examples and RDBMS may be used to teach database systems. For example, the natural language generation framework should be generalizable. Ideally we would like to generate natural language descriptions of QEPs using one application-specific dataset (*e.g.,* movies) and then use it for other applications (*e.g.,* hospital) on any off-the-shelf RDBMS.
- **Psychology-awareness.** Any technology-enabled learning framework has to be *learner-centric*, *i.e.,* it has to be cognizant of the psychology of learners. Any deployable solution has to be palatable and engaging to learners so that they are motivated to learn and explore. Hence, these solutions need to be consistent with various cognitive psychology and motivation theories to have practical impact. For example, the NL descriptions for different queries must not use the same language to describe various operations in QEPs. Similarly, highly similar AQPs should not be exposed to the learners. Otherwise, learners may feel bored after viewing several AQPs or reading the NL descriptions for several queries. In fact, this is consistent with research in psychology that have found that repetition of messages can lead to annoyance and boredom [12] resulting in purposeful avoidance [22], content blindness [23], and even lower motivation [38].

3.8 Towards Data-driven Education

As remarked in Section 1, learning can be facilitated by education. Hence, technological platforms that address the aforementioned challenges may pave the way for *data-driven* education due to rich access to *interaction log* data of learners. Such log data may consists of access times of learners, history of queries formulated by learners, temporal information related to various interactions, among others. This provides a rich data source for building data-driven techniques to facilitate education by analyzing these data at both individual and group levels and correlating them with the performances of learners in tests (*i.e.,* academic outcomes). A non-exhaustive list of questions that can be answered by exploiting the log data to facilitate data-driven education is as follows:

- How do the type and complexity of SQL queries posed by learners evolve over time? What are the activity

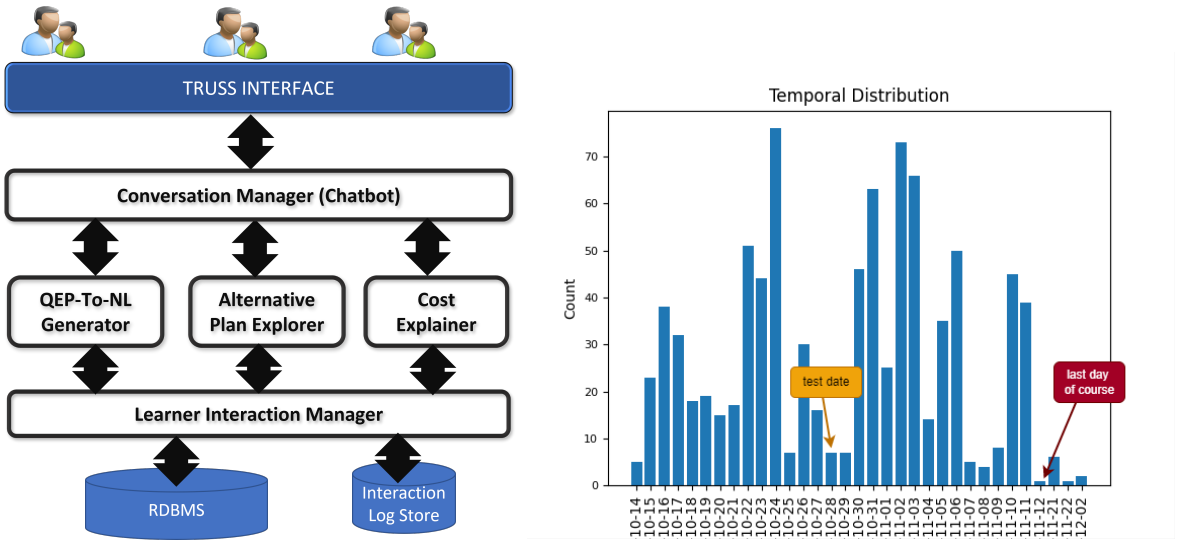


Figure 4: (a) Architecture of TRUSS (left); (b) No. of queries versus time (right).

patterns of learners during a semester? Answers to these questions may provide insights on motivation and learning habits of learners.

- How do learners learn relational query processing? Numerous studies in cognitive psychology show that *spacing* (*i.e.,* distributing practice over more sessions) significantly improves long-term learning compared to *massing* (*i.e.,* practice in longer sessions) [9, 10, 39, 41]. The interaction data may enable us to build models to predict learners demonstrating massing, thereby enabling timely intervention to nudge them to more effective learning habits.
- Research in education posits that technology can be used effectively as a short but focused intervention to improve learning especially when there is regular and frequent usage over a period of several weeks [24]. In our context, it is expected that the platforms are also for focused usage over a period of few weeks. Do they help learners to perform better in tests and coursework? Is there any correlation between frequency of engagement with a platform and performance? Answer to this may provide data-driven insights to the effectiveness of these tools in learning.
- Do learners continue to use the platforms even after the end of a database course? This may indicate intrinsic motivation to learn relational query processing.
- Can the queries posed by learners over time shed light on the difficulties they face with respect to the learning and understanding of relational query processing and optimization? Answer to this question may facilitate the design of more effective and efficient pedagogical strategies to improve effectiveness of teaching.

In summary, addressing the aforementioned research issues provide us a unique opportunity to take a data-driven approach to the education of relational query processing that may otherwise be infeasible through traditional mode of teaching.

4 The TRUSS System

Figure 4(a) depicts the high-level architecture of the TRUSS (**T**echnology-enabled Lea**R**ning of Q**U**ery Proce**SS**ing) system that we are currently building to address the challenges introduced in the preceding section. The *QEP-to-NL Generator*, *Alternative Plan Explorer*, and *Cost Explainer* components aim to address the challenges in

Sections 3.2, 3.3 & 3.4, and 3.5, respectively. The *Conversation Manager* is to realize the unifying chatbot (Section 3.6) and the *Learner Interaction Manager* is responsible to facilitate data-driven education (Section 3.8). In this section, we briefly describe our recent efforts to build three frameworks, NEURON [31] and LANTERN [14], that aim to address the problem described in Section 3.2 (*i.e., QEP-to-NL Generator*), and MOCHA [40] that takes an initial step to address the problem in Section 3.3 (*i.e., Alternative Plan Explorer*). The reader may refer to [14, 31, 40, 42] for details on these frameworks.

4.1 NEURON and LANTERN

NEURON [5, 31] is the first system that exploits a rule-based interpretation engine to generate NL description for QEP in PostgreSQL. Specifically, given the QEP of a SQL query, NEURON first parses and transforms the QEP of an SQL query into an operator tree where each node contains relevant information associated with a plan (*e.g.,* filter conditions). Next, it traverses the tree and generates a NL description of the node based on NL templates and the information it carries. It also supports a preliminary *natural language question answering* system that allows a user to seek answers to a variety of concepts and features associated with a QEP.

NEURON is a rule-based framework that is tightly integrated with PostgreSQL. Hence, it is not generalizable (Section 3.7). LANTERN [3, 14, 42] addresses this limitation by not only making the solution generalizable but also psychology-aware. It incorporates a *declarative framework* called POOL to empower *subject matter experts* (SMEs) create and manipulate the NL descriptions (*i.e.,* labels) of physical operators, which are the building blocks of QEPs. The data definition in POOL allows one to declaratively create physical operator objects associated with a specific RDBMS. For example, one can create the definition of hash join operator in PostgreSQL (pg) as follows.

```
CREATE POPERATOR hashjoin FOR pg
(ALIAS = null,
TYPE = 'binary',
DEFN = null,
DESC = 'perform hash join',
COND = 'true',
TARGET = null)
```

In particular, the TYPE attribute can take either 'unary' or 'binary' value. The DESC attribute allows one to specify a natural language description of the operation performed by the operator. The COND attribute takes a Boolean value to indicate whether a specified condition (*e.g.,* join condition) should be appended to the natural language description of an operator. Values of all attributes are taken from the atomic type `string` (possibly empty). Note that no relation or condition is specified in DESC. This is because these are added automatically to DESC by exploiting TYPE and COND attributes of an operator. For instance, since TYPE is 'binary' in the above definition, two variables representing join relations will be added automatically to the description of `hashjoin`. Lastly, the TARGET attribute allows one to specify the operator name which is supported by the defined operator. For example, TARGET is set to 'hash join' for the definition of `hash` operator.

The key goals of the data manipulation component of POOL are to provide syntactical means to support (a) retrieval of specific properties (*i.e.,* attributes) of physical operators using SQL-like SELECT-FROM-WHERE syntax, (b) generation of the *template* for natural language description of an operator using the COMPOSE clause, and (c) update properties of physical operator objects using UPDATE and REPLACE clauses. Specifically, the COMPOSE clause uses the *desc*, *type*, and *cond* attributes of operators to generate the template. For example, the template generation for the `hash` operator can be specified as follows.

```
COMPOSE hash FROM pg
```

The above statement will return the template *"hash $\$R_1\$$"*, which can be subsequently used by LANTERN to generate specific description of the `hash` operator in a QEP. Also, observe that R_1 is appended based on the *type*

attribute of the hash object. An example to generate the NL description template of the hash join operator is as follows.

```
COMPOSE hash, hashjoin FROM pg
USING hashjoin.desc = 'perform hash join'
```

The above statement generates the following template: *"hash $\$R_1\$$ and perform hash join on $\$R_2\$$ and $\$R_1\$$ on condition $\$cond\$$"*.

The update statement can be exploited to assign definition or description of an operator from one commercial database to another, thereby making it more efficient for an SME to specify properties of physical operators. The following example demonstrates how the description of hash join in PostgreSQL is transferred to the hash join operator in DB2.

```
UPDATE db2
SET desc = (SELECT desc
 FROM pg WHERE pg.name = 'hashjoin')
WHERE db2.name = 'hsjoin'
```

It can also be used along with the REPLACE clause to transfer definition or description of an operator object to another within the *same* source. For example, one can transfer the description of hash join to nested loop join by replacing the word 'hash' with 'nested loop' as follows.

```
UPDATE pg
SET desc = REPLACE((SELECT desc FROM pg AS pg2
WHERE pg2.name = 'hashjoin'), 'hash', 'nested loop')
WHERE pg.name = 'nested loop join'
```

Note that the REPLACE clause takes three parameters as input, namely, the description or definition of an operator object, the string in it that needs to be replaced (*e.g.,* 'hash'), and its new replacement string (*e.g.,* 'nested loop').

Once the physical operator objects for different RDBMS are created in POOL and stored, the physical operator tree of a given query in any RDBMS can be augmented by automatically annotating relevant nodes with NL descriptions by leveraging the COMPOSE statement and replacing the place holders in NL templates with specific relations, attribute names, and predicates relevant to the query. The NL description generation framework then utilizes this augmented operator tree and *integrates* a rule-based and deep learning-based techniques. In particular, the latter infuses language variability in the descriptions opportunely. This strategy has been shown to mitigate the impact of boredom on learners that may arise due to repetitive statements in different NL descriptions [42].

4.2 MOCHA

MOCHA (iMpact of Operator CHoices visuAlizer) [4, 40] aids learner-friendly interaction and visualization of the impact of alternative physical operator choices on a selected QEP for a given SQL query. It is built on top of PostgreSQL. Given an SQL query and learner-specified *operator preferences* (*e.g.,* merge join, index scan), MOCHA automatically visualizes the impact of these choices on the selected QEP. Specifically, it exploits the *planner method configuration*[1] feature of PostgreSQL to generate AQPs based on a user input. The configuration parameters in this feature provide a way to enforce the query optimizer to choose a query plan with certain user-specified physical operators. By default, all parameters are turned on during query processing. A query request is sent to PostgreSQL using the default settings to retrieve the QEP of a query.

[1] www.postgresql.org/docs/9.2/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS.

In order to retrieve AQPs, a learner may select a subset of the configuration parameters (through a user-friendly visual interface) based on the physical operators that she intend to view in these plans. In this case, the corresponding parameters are set to "true" (*e.g.,* `SET enable_mergejoin = true`) in the query request. MOCHA supports two modes for generating alternative plans, namely, *single mode* and *multiple mode*. In the former mode, MOCHA sends a query request to PostgreSQL in which the *unselected* parameters are set to "false" to generate an AQP containing the operators corresponding to the selected parameters that are relevant to the processing of the query. In the latter mode, every selected parameter is either set to "true" or "false" to create all possible combinations of these parameters. MOCHA iterates through these combinations and sends corresponding query requests to PostgreSQL. It only maintains all *distinct* plans retrieved from these requests. To facilitate learning, it provides a learner-friendly GUI to detect and visualize various structural and cost differences between a selected AQP and the QEP.

MOCHA also generates a natural language-based explanation that goes beyond the conventional least-cost-based explanation to connect established knowledge related to usage scenarios of different physical operators that a learner has learnt from textbooks with the operators in a QEP. The current version manually extracts usage scenarios of different physical operators from the relevant literature. This is feasible since there is a small number of physical operators in PostgreSQL. Then a set of documents containing these usage scenarios is indexed using an inverted index where each document is associated with a single physical operator. For a given QEP, it identifies relevant operators and retrieves associated predicates and join conditions, if any. The text explanation is then generated for an operator by utilizing a rule-based template, the inverted index to retrieve corresponding usage scenario, and database statistics information (*e.g.,* selectivity). The generated explanation is visually displayed on the visual interface of MOCHA. For example, an explanation could be *"the* QEP *uses index scan on the* `lineitem` *table as it is faster due to the high selectivity of the predicate (*i.e., `l_orderkey = orders.o_orderkey)"*.

In the future, we intend to generalize MOCHA to accommodate major RDBMS, support visualization of the impact of join ordering, and automate the manual extraction of usage information of various physical operators in these RDBMS. More importantly, we wish to deploy MOCHA in our learning environment and investigate its impact on students taking the database systems course.

4.3 Usage and Impact of NEURON

NEURON and LANTERN are currently deployed in database systems courses in NTU and Xidian University. We now briefly describe our initial efforts to measure NEURON's impact on the learning of QEP. To this end, we introduced it to students taking the undergraduate database systems course (*CZ4031*) in NTU in the August semester of 2021. 166 students were enrolled in this course. These students are pursuing a variety of degrees such as computer science, computer engineering, data science and analytics, and business and computing. In particular, the topic of query processing and optimization was covered in 4 weeks (September-October) over eight 1-hour lectures. On October 28th, the students took a test on the topic of query processing and optimization. The following message was sent to the students on 14th October, 2021: *"If you wish to understand query execution plans (*QEP*) generated by PostgreSQL for different* SQL *queries, you may use the software called* NEURON *at* `https://neuron.scse.ntu.edu.sg/`. NEURON *translates the* QEP *of a query to natural language description."* We did not nudge students any further on using NEURON or give them any hints on whether questions related to natural language descriptions of query plans will appear in the test. Hence, students were not aware of any assessment-related rewards if they used the tool. The goal here is to observe whether learners will use it without any nudging or grades-related rewards and whether they will benefit from it.

We observe the usage of NEURON from 14th October to 2nd December by analyzing the log file. Note that 12th November was the last date of the course culminating with the submission of a course project. Since a user needs to access NEURON by logging using an email address, we were able to match majority of the email addresses that accessed it with those registered for the course. There were 69 distinct learners (41.5%) using NEURON during this period. Figure 4(b) reports the number of queries posed by learners over time. In total, 888

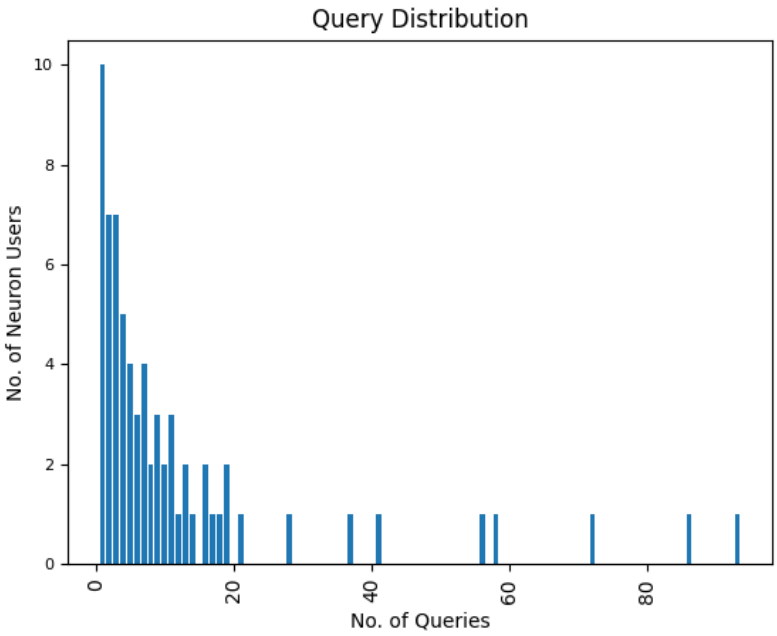


Figure 5: No. of queries versus no. of users.

valid SQL queries (484 distinct queries) were executed on NEURON. The distribution of the number of queries versus the number of distinct learners who posed that number of queries is shown in Figure 5. Observe that more than 85% of them posed more than one query and the maximum number of queries posed by a single user is 93! Prior to October 28th (test date), the number of distinct users is 48 and the number of queries posed is 409 (211 distinct queries). Hence, the usage of NEURON continued even after the test as learners may be using it to understand QEPs for their project work. Some students used it even after the official end date of the course probably demonstrating intrinsic motivation to explore QEPs.

To investigate whether NEURON may benefit learners to understand QEP, we use the test as a proxy. In the test, a question was specifically set to this end. The students were asked to explain in natural language the visual format of a QEP (in PostgreSQL) for an SQL query on IMDb database involving two joins, scans, and sort operations. The question carried 10 marks. Observe that it matches very closely to NEURON's goal. This enables us to evaluate more accurately possible impact of NEURON on the test performance.

In order to avoid any bias, a teaching assistant (TA) who is not involved with NEURON graded the answers to this question. The TA was given the solution and was allowed to set the marking scheme for the question. The number of students who took the test is 162. The average score of students who used (resp. not used) NEURON prior to test is 8.43 (resp. 7.07). The maximum, minimum, and median scores of these two groups are (10, 6.5, 8) and (10, 0, 7.5), respectively. Among the non-NEURON users, the percentage of students with scores lower than the minimum score in the NEURON user group (*i.e.,* 6.5) is 21.31%. Furthermore, the percentage of NEURON users (resp. non-NEURON users) with scores higher than the average score of 8 is 47.5% (resp. 35.25%). Some of the common errors made by students are (a) not describing in natural language; (b) incorrect sequence of steps; (c) not including filter conditions in the scan operations; and (d) unclear specifications of operators and intermediate results. Observe that these errors could have been mitigated with the usage of NEURON.

In summary, while we cannot infer causal link from the initial results, it is possible NEURON may improve learning of query execution strategies. We are still in the early stages of understanding the impact of this tool on learning. We intend to use NEURON and LANTERN in future semesters to gather sufficient longitudinal data for a

more detailed investigation of technology-enabled learning and their impact on data-driven education.

5 Conclusions

Impact of digital technologies on learning has consistently shown positive benefits when they are aligned. With the advent of data science and lifelong learning, there has been growing interest in the database course from adult learners with diverse background. This necessitates us to revisit the traditional way we teach this course by supplementing it with technological support to improve learning. This paper contributes a vision of technology-enabled learning of the topic of relational query processing in a database systems course. Specifically, our vision attempts to carve out a substantially new research topic that is at the intersection of learning sciences and data management to improve learning of relational query processing. To the best of our knowledge, this vision has not been systematically investigated before, prior to our recent publications.

Measures of success. Successful realisation of this vision will improve learning and understanding of the complex topic of relational query processing. But several non-trivial and novel research challenges as articulated in the paper need to be overcome to realize it. Adoption of the potential solutions by real-world learners as a supplement to traditional modes of learning will be another measure of success.

Wider applicability. We focused on the topic of relational query processing since it is one of the most challenging topic in a database systems course. Nevertheless, it is easy to see that our vision of technology-enabled learning can be extended to other topics such as enabling technologies to facilitate learning of SQL queries.

References

[1] The IMDb database. `https://relational.fit.cvut.cz/dataset/IMDb`.

[2] Advanced query tuning concepts. `https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms191426(v=sql.105)?redirectedfrom=MSDN`, 2012.

[3] LANTERN software. `https://howardlee.cn/lantern/`.

[4] MOCHA software. `https://howardlee.cn/mocha/`.

[5] NEURON software. `https://howardlee.cn/#/`.

[6] USESCO Insititute of Lifelong Learning. UNESCO Global Network of Learning Cities. Accessible at `https://uil.unesco.org/lifelong-learning/learning-cities`, 2019.

[7] E. M. Anderman, H. Patrick. Achievement goal theory, conceptualization of ability/intelligence, and classroom climate. *In Handbook of research on student engagement*, Springer: Boston, MA, 2012.

[8] A. Bhangdiya, B. Chandra, B. Kar, B. Radhakrishnan, K. V. M. Reddy, S. Shah, S. Sudarshan. The XDa-TA system for automated grading of SQL query assignments. *In ICDE*, 2015.

[9] E. L. Bjork, R. Bjork. Making things hard on yourself, but in a good way. *Psychology in the Real World*, 59-68, 2011.

[10] R. A. Bjork, J. Dunlosky, N. Kornell. Self-regulated learning: Beliefs, techniques, and illusions. *Annual review of psychology*, 64, 417–444, 2013.

[11] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, C. Fraser. Smooth scan: robust access path selection without cardinality estimation. *The VLDB Journal*, 27(4):521-545, 2018.

[12] J. T. Cacioppo and R. E. Petty. Effects of Message Repetition and Position on Cognitive Response, Recall, and Persuasion. *Journal of Personality and Social Psychology* ,37, 1: 97-109, 1979.

[13] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. *In PODS*, 1998.

[14] P. Chen, H. Li, S. S. Bhowmick, S. R. Joty, W. Wang. LANTERN: Boredom-conscious Natural Language Description Generation of Query Execution Plans for Database Education. *In SIGMOD*, 2022.

[15] J. Danaparamita, W. Gatterbauer. QueryViz: Helping Users Understand SQL Queries and Their Patterns. *In EDBT*, 2011.

[16] E. L. Deci, R. J. Vallerand, L. G. Pelletier, R. M. Ryan. Motivation and education: The self-determination perspective. Educational psychologist, 26(3-4), 325-346, 1991.

[17] H. Garcia-Molina, J. D. Ullman, J. Widom. Database Systems: The Complete Book. *Prentice-Hall*, 2002.

[18] M. Gawade, M. L. Kersten. Stethoscope: A platform for interactive visual analysis of query execution plans . *PVLDB*, 5(12), 2012.

[19] R. Gross. Psychology: The Science of Mind and Behaviour. *Hachette UK*, ISBN 978-1-4441-6436-7.

[20] J. Gao, M. Galley, L. Li. Neural Approaches to Conversational AI. *In ACL*, 2018.

[21] J. R. Haritsa. The Picasso Database Query Optimizer Visualizer. *In PVLDB*, 3(2), 2010.

[22] M. R. Hastall and S. Knobloch-Westerwick. Severity, Efficacy, and Evidence Type as Determinants of Health Message Exposure. *Health Communication*, 28, 4: 378-388, 2013.

[23] G. Hervet, K. Guerard, S. Tremblay, M. Saber Chtourou. Is Banner Blindness Genuine? Eye Tracking Internet Text Advertising. *Applied Cognitive Psychology*, 25, 5: 708-716, 2011.

[24] S. Higgins, Z. M. Xiao, M. Katsipatak. The Impact of Digital Technology on Learning: A Summary for the Education. *Education Endowment Foundation*, 2012.

[25] Y. Hu, Z. Miao, Z. Leong, H. Lim, Z. Zheng, S. Roy, K. Stephens-Martinez, J. Yang. I-Rex: An Interactive Relational Query Debugger for SQL. *In ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2022.

[26] Z. Ives, J. Gehrke, J. Giceva, A. Kumar, R. Pottinger. VLDB Panel Summary: "The Future of Data(base) Education: Is the Cow Book Dead?". *SIGMOD Rec.*, 50(3), 2021.

[27] A. E. Kazdin. Motivation: an overview. *Encyclopedia of Psychology*, American Psychological Association, ISBN 978-1-55798-187-5, 2000.

[28] H. Kim, B.-H. So, W.-S. Han, H. Lee. Natural Language to SQL: Where Are We Today? *PVLDB*, 13(10), 2020.

[29] A. Kokkalis, P. Vagenas, A. Zervakis, A. Simitsis, G. Koutrika, Y. E. Ioannidis. Logos: A System for Translating Queries into Narratives. *In SIGMOD*, 2012.

[30] A. Leventidis, J. Zhang, C. Dunne, W. Gatterbauer, H. V. Jagadish, M. Riedewald. QueryVis: Logic-based Diagrams help Users Understand Complicated SQL Queries Faster. *In SIGMOD*, 2020.

[31] S. Liu, S. S. Bhowmick, W. Zhang, S. Wang, W. Huang, S. Joty. NEURON: Query Optimization Meets Natural Language Processing For Augmenting Database Education. *In SIGMOD*, 2019.

[32] Z. Miao, S. Roy, J. Yang. Explaining Wrong Queries Using Small Examples. *In SIGMOD*, 2019.

[33] D. Miedema, G. Fletcher. SQLVis: Visual Query Representations for Supporting SQL Learners. *In VL/HCC*, 2021.

[34] D. Miedema, E. Aivaloglou, G. Fletcher. Identifying SQL Misconceptions of Novices: Findings from a Think-Aloud Study. *In ICER*, 2021.

[35] J. Nakamura, M. Csikszentmihalyi. Flow theory and research. *Oxford Handbook of Positive Psychology*, Oxford University Press, 2009.

[36] R. Ramakrishna, J. Gehrke. Database Management Systems. *McGraw-Hill, Inc.*, USA, 2020.

[37] R. M. Ryan, E. L. Deci. Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary educational psychology*, 25(1), 54-67, 2000.

[38] D. W. Schumann, R. E. Petty, D. S. Clemons. Predicting the Effectiveness of Different Strategies of Advertising Variation: A Test of the Repetition-Variation Hypotheses. *Journal of Consumer Research*, 17, 2: 192, 1990.

[39] N. C. Soderstrom, R. A. Bjork. Learning versus performance: An integrative review. *Perspectives on Psychological Science*, 10(2), 176-199, 2015.

[40] J. Tan, D. Yeo, R. Neoh, H.-E. Chua, S. S. Bhowmick. MOCHA: A Tool for Visualizing Impact of Operator Choices in Query Execution Plans for Database Education. *PVLDB*, 15(12), 2022.

[41] K. Taylor, D. Rohrer. The effects of interleaved practice. *Applied Cognitive Psychology*, 24(6), 2010.

[42] W. Wang, S. S. Bhowmick, H. Li, S. Joty, S. Liu, P. Chen. Towards Enhancing Database Education: Natural Language Generation Meets Query Execution Plans. *In SIGMOD*, 2021.

[43] A. Wigfield, J. S. Eccles. Expectancy-value theory of achievement motivation. *Contemporary educational psychology*, 25(1), 68-81, 2000.

[44] C. Zhu, C. Ji, Y. Zhang, X. Liu, A. Jatowt, S. S. Bhowmick, C. Sun, T. Zhao. Towards Automatic Support for Leading Court Debates: A Novel Task Proposal & Effective Approach of Judicial Question Generation. *To Appear in Neural Computing and Applications (NCAA)*, Springer, 2022.

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:

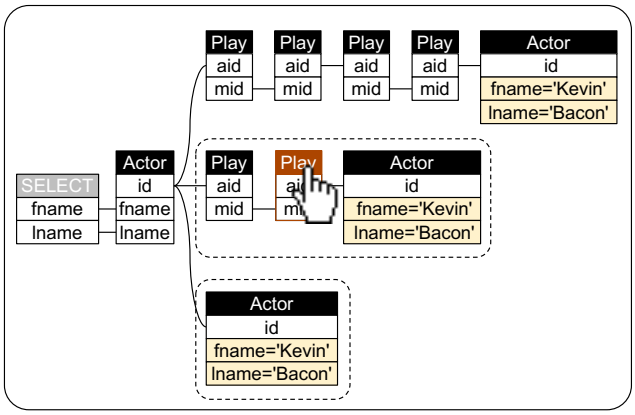
Scenario 1 (Data scientists reusing past queries): A group of data analysts are collaboratively analyzing movie data. This data is stored in a shared data repository. In addition to the data itself, they are also sharing their queries using a Collaborative Query Management System (CQMS) [4, 5, 16, 24, 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

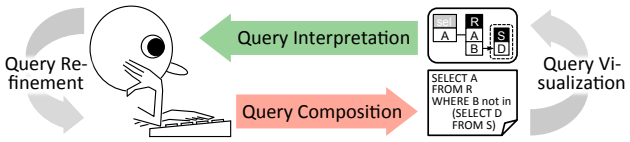


Figure 2: The goal of query visualization is to complement (but not substitute) the composition of queries by creating automatic visualizations of queries. Composition of a query is still performed via unambiguous and expressive text. The transformation from text into a visualization can abstract away 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.

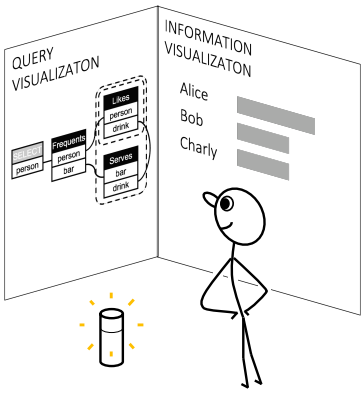


Figure 3: Scenario 5: An analyst dictates queries to her voice assistant which then shows the query as understood together with the query answers.

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.

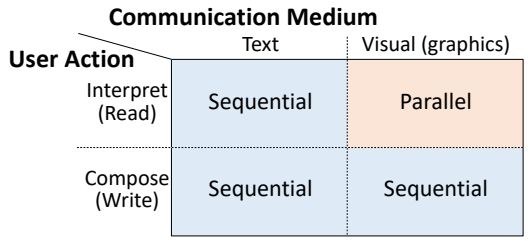


Figure 4: *Composing* a query with a visual query language is as sequential as composing it with SQL. *Interpreting* a visualization (whether of information or a query) is the only modus in which a user can act on information in parallel, leveraging the speed of the human perceptual system (orange =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

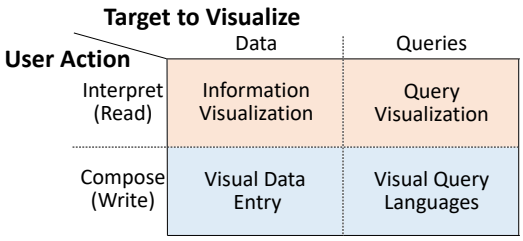


Figure 5: *Visual Query Languages* allow a user to compose queries. They have been widely studied and have a rich history. In contrast, *Query Visualization* helps the user understand an existing query just as *Information Visualization* helps understand data (orange = easier, blue = harder).

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

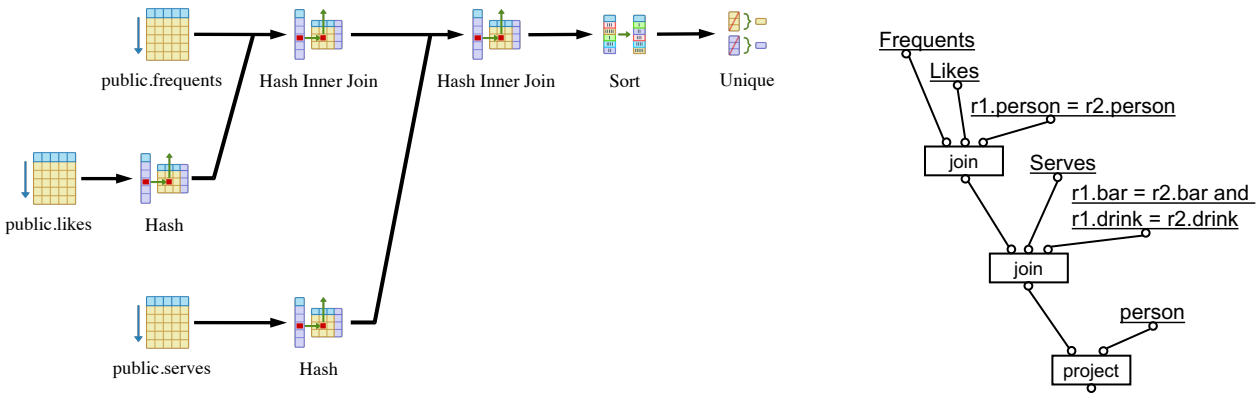
There are three main alternative approaches for helping users understand existing queries:

(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

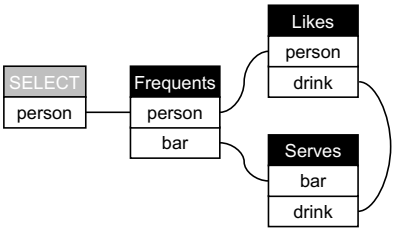
The challenge of query visualization is to find appropriate visual metaphors that (i) allow users to quickly understand a query's intent, even for complex queries, (ii) can be easily learned by users, and (iii) can be obtained from SQL by automatic translation, including a visually-appealing automatic 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

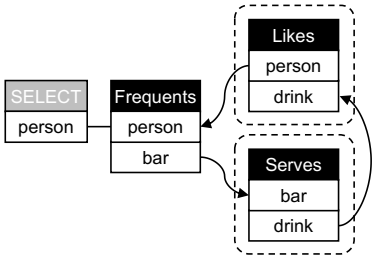


(b) Q_{some} in QueryVis

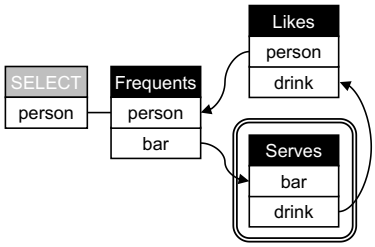
Figure 7: Principles 1 & 2: Visualizing a conjunctive query should follow a familiar UML notation: *Find persons who frequent some bar that serves some drink they like*. The only novelty is a dedicated output table on the left, emphasizing the compositionality of the relational model, and supporting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

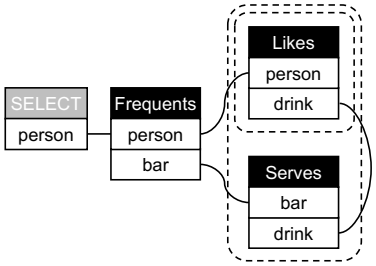
(a) Q_{only}



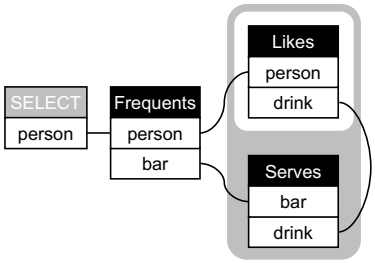
(b) Q_{only}



(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:

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

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).

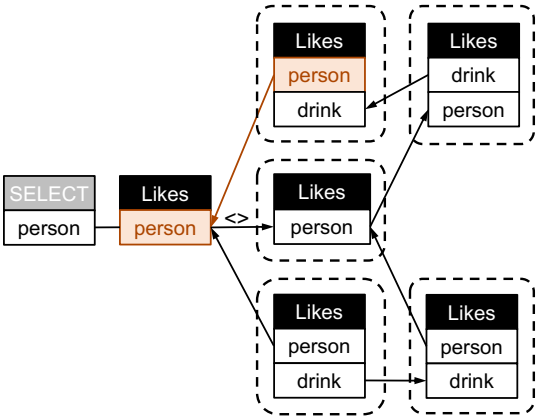
(4) Expose (and not hide) relational patterns: We believe that a query visualization should expose the relational pattern used in a textual query, instead of replacing it with an abstraction and concepts that go beyond the relational model. This requires a visualization to use the same number of input 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).

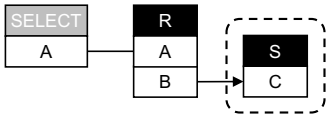
(8) Logic-based visual transformations: Nested negated quantifiers are particularly difficult to understand for users [79, 80]. Simple visualizations of logical transformation can further help show the query in a more

```
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

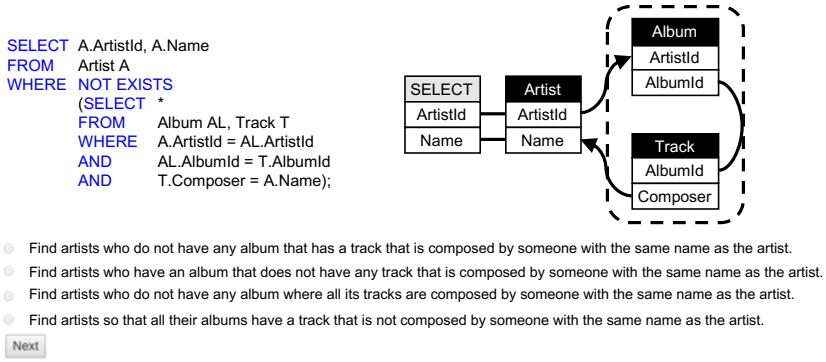


Figure 11: Section 4.1: Example query from our user study. The query is shown in the *Both* condition, in which a participant sees the query in both SQL (left) and our QueryVis diagram (right).

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

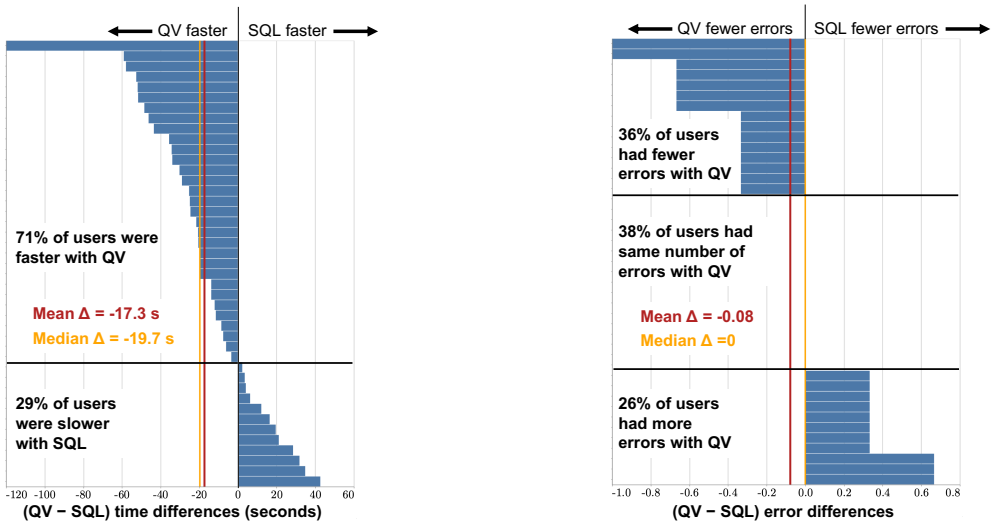


Figure 12: Section 4.1: Distribution of (QV−SQL) time and error differences for each participant on 9 MCQs.

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.

Structured Data Representation in Natural Language Interfaces

Yutong Shao, Arun Kumar, and Ndapa Nakashole
University of California, San Diego, USA
{yshao,arunkk,nnakashole}@eng.ucsd.edu

Abstract

A Natural Language Interface (NLI) enables the use of human languages to interact with computer systems, including smart phones and robots. Compared to other types of interfaces, such as command line interfaces (CLIs) or graphical user interfaces (GUIs), NLIs stand to enable more people to have access to functionality behind databases or APIs as they only require knowledge of natural languages. Many NLI applications involve structured data for the domain (e.g., applications such as hotel booking, product search, and factual question answering.) Thus, to fully process user questions, in addition to natural language comprehension, understanding of structured data is also crucial for the model. In this paper, we study neural network methods for building Natural Language Interfaces (NLIs) with a focus on learning structure data representations that can generalize to novel data sources and schemata not seen at training time. Specifically, we review two tasks related to natural language interfaces: i) semantic parsing where we focus on text-to-SQL for database access, and ii) task-oriented dialog systems for API access. We survey representative methods for text-to-SQL and task-oriented dialog tasks, focusing on representing and incorporating structured data. Lastly, we present two of our original studies on structured data representation methods for NLIs to enable access to i) databases, and ii) visualization APIs.

1 Introduction

Natural Language Interfaces (NLIs) seek to enable user-system interactions using natural language. Compared to other types of interfaces, such as command-line or graphic interfaces, NLIs have a much lower learning curve. They can be used by lay users with little to no extra training, thus attractive in practice.

NLIs have been an active research for decades [1, 2]. Early methods had limited success, possibly due to the absence of powerful and effective models for language understanding. Recently, with the rise of deep learning models, especially Transformer-based models in NLP [46], the overall performance on a variety of language understanding tasks, including NLI-related tasks, has seen a significant boost. Nonetheless, the state-of-the-art (SOTA) NLI models are still far from perfect, and are do not yet reach the level of being fully deployable in real products. In this work, we investigate the progress and challenges of NLI-related tasks. Specifically, we focus on two types of tasks: *semantic parsing* and *task-oriented dialogs (TOD)*. In addition, we focus on the understanding and representation of *structured data*, such as knowledge bases (KBs), databases (DBs) or tables, in NLI-related tasks. In practice, NLI applications inevitably need to incorporate and understand the relevant "backend" structured data for the domain (hotel booking, product searching, factual question answering, etc.)

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Thus, besides language comprehension, a correct understanding of structured data is also crucial for the model to make the correction decisions.

In following sections, we first provide a more detailed descriptions of the NLI-related tasks we study. For each task, we survey representative methods for the tasks, especially regarding structured data representation; then we introduce our own work related to the task. A road map of this paper is illustrated in Figure 1.

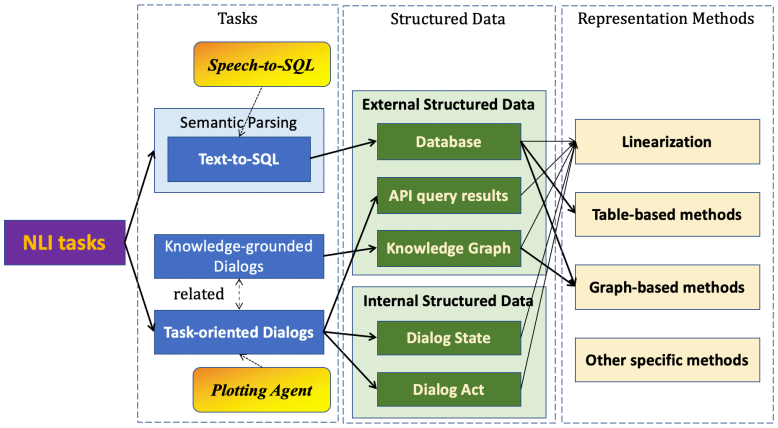


Figure 1: The road map of the paper. We discuss several NLI-related tasks, relevant structured data and corresponding representation methods. We introduce two of our own related work, Speech-to-SQL and Plotting Agent, under the topic.

2 Preliminaries

2.1 Popular NLP models

We briefly introduce several model architectures that are widely used today in the NLP community and will be frequently mentioned in this paper.

Sequence-to-sequence (S2S) A sequence-to-sequence (S2S) model aims to map an input sequence to an output sequence [41]. In NLP, the input and output sequences are usually natural language sentences. Many NLP tasks can be modeled by S2S models, such as machine translation, summarization, and dialog.

S2S models are mostly based on an *encoder-decoder* model architecture, in which the *encoder* aims to obtain an intermediate vector or tensor representation that includes necessary information from the input sequence (named the *encoding* of input), and the *decoder* generates the output sentence based on the input encoding. Both encoder and decoder can be modeled by recurrent neural networks (RNNs), such as LSTM or GRU [42, 43]; or by transformer models, which we briefly introduce below.

In the basic version of RNN-based S2S model architecture,the encoder absorbs an input sentence and encodes it into a single *encoding vector*; the decoder takes the encoding vector to generate an output sentence as a conditioned language model (i.e. generate tokens in auto-regressive manner, based on previous tokens). Intuitively, the encoding vector is an information bottleneck in the architecture because all information from the input has to be incorporated into a single vector.

Attention The attention mechanism [44, 45] was proposed to overcome the above-mentioned bottleneck. It allows the decoder states to "look through" the encoding vector and directly interact with encoder states. In detail,

given a decoder state vector d and encoder state vectors $e_{1:N}$, we compute an *attention context vector* c as follow:

$$\alpha_k = \frac{\texttt{sim}(d, e_k)}{\sum_{i=1}^{N} \texttt{sim}(d, e_i)}; c = \sum_{k=1}^{N} \alpha_k e_k$$

where α_k are called *attention weights*, and `sim` is a function measuring vector similarities, such as dot product ($a^T b$), bilinear product ($a^T W b$), linear combination ($w_1^T a + w_2^T b$), etc. The attention context vector is then added or concatenated with the decoder state vector before predicting the output token.

Self-attention and Transformers The Transformer architecture, proposed in [46], fully replaces RNNs with *self-attention* layers. Self-attention layers encode a sequence by applying attention from the representation of each token to all other tokens in the sequence, and use the attention context vector (with residual connection, i.e. summed with the pre-attention representation vector) as the output token representation. Transformers are shown to be particularly useful for pretrained language models (PLMs), i.e. first *pretrain* the model on an extremely large corpus with self-supervision, and then *fine-tune* the pretrained parameters on a downstream task [47, 48].

2.2 NLI-related Tasks

The topic of NLIs is broad. In this work, we focus on two representative tasks: semantic parsing and task-oriented dialogs (TOD). These tasks are useful in practice and widely studied, especially in the current era of deep learning. We briefly introduce the tasks in this section and take in-depth investigations in the following sections.

Semantic Parsing Semantic parsing aims to map a natural language sentence into a structured, executable logical form to represent the semantics of the sentence. It can be seen as a straightforward formulation of an NLI: parsing the natural language query into system-understandable form. There are different formats of such executable logical forms, such as GeoQuery [3], lambda-DCS [4], SparQL (for knowledge graphs (KG)) and SQL (for relational databases (DB)).

Task-oriented Dialog (TOD) Task-oriented dialog (TOD) systems interact with users to complete certain tasks. The task objectives are usually in the form of making API calls to obtain information (e.g. look for restaurants) or take actions (e.g. make a restaurant reservation) [23–26]. From a practical perspective, for complex tasks, it is hard for user requests to be described and completed in a single utterance. Therefore, interactions are necessary between the user and system to complete the task, which makes a task-oriented dialog.

3 Semantic Parsing

3.1 Introduction

Early work in semantic parsing are based on grammar rules induction and feature-based rule selection [3, 5]. Since the rise of deep learning, parsers are mostly based on neural models with encoder-decoder architectures that generate the output logical form directly from input text [6–8]. Moreover, recent work pay increasingly more attention on the *generalization ability* of trained models, especially on generalizing over changes in the backend structured data. The advantages of such generalization is to alleviate the burden of re-collecting in-domain data and re-training the model whenever we update our DB or KG. In order to generalize to unseen structured data, we have to incorporate the structured data as part of the input to the model, instead of fully include them into the model weights. It thus raises the problem of *structured data representation*, which is basically on how to effectively incorporate and leverage the structured data for a better model performance.

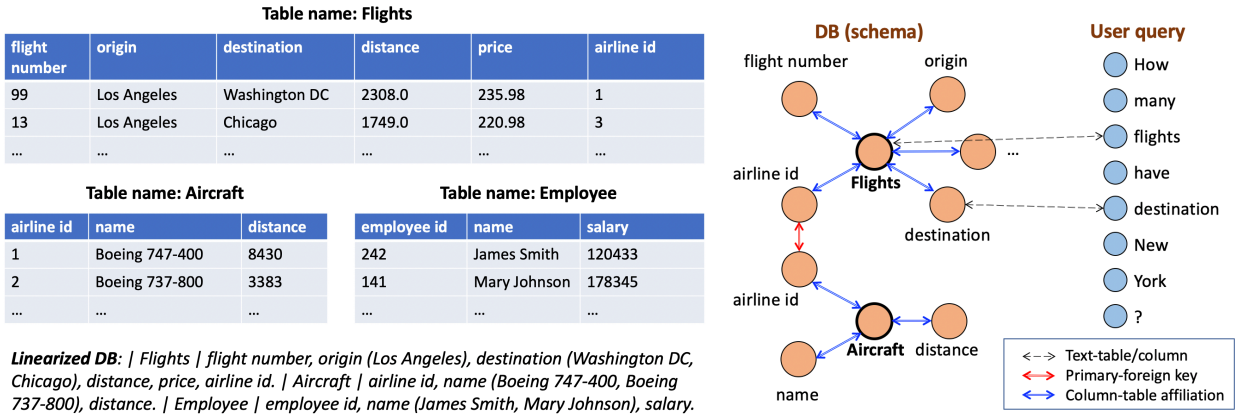
In what follows, we focus on structured data representation on semantic parsing with SQL as the logical form, which is also called text-to-SQL. As mentioned, the task of text-to-SQL takes as input the user utterance in natural language and the backend DB. A recent benchmark on text-to-SQL, Spider [9], explicitly tests the generalization ability to unseen DBs by separating the set of DBs used in train/dev/test splits of the benchmark. It provides a good testbed for modern text-to-SQL methods and has been widely used by previous work in this direction.

3.2 Structured Data Representation in Text-to-SQL

3.2.1 Basic Method: Linearization

In the task of text-to-SQL, the structured data to be incorporated are the DBs. The most straightforward idea is to *linearize* the DB, i.e. transform it into a token sequence, and concatenate it with the user question to form the input text. A simple example is shown in Figure 2a. By linearizing, the structured data is mapped into unstructured text modality, thus can fit into regular text-processing models.

Despite of its simplicity, linearization has shown to be a very useful way to represent DBs [12, 13]. There are also work focusing on injecting more useful information into such linear representations. For example, in [14] the authors propose to find *anchor text* which are input text tokens matching DB cell values, and add them to the linearized DB, next to the column of the cell (similar to the format of cell inclusion shown in Figure 2a). Also, DB linearization works well with large-scale pretraining [15, 16]. This is possibly due to the compatibility of linearized DB and unstructured text, thus effective pretraining objectives (such as masked language-modeling (MLM)) can be adopted.



(a) DB linearization.

(b) DB graph.

Figure 2: DB linearization and graph illustration. (a) An example of DB linearization. Notice that the included cell values are deliberately selected to showcase the idea; in practice, cells are not always available, depending on the dataset, and cell value selection is non-trivial and method-specific. (b) An example of DB graph. Some columns, tables and cells are omitted for visualization clarity. In detail, the design of relation set is also method-specific. This figure only illustrates the high-level idea.

3.2.2 Table Representation

Essentially, a DB is made up by tables; therefore, we can exploit the tabular structure to better represent the information. A lot of exisiting work target table representation, not specific for text-to-SQL but generally for any task involving tables (other tasks include table QA, table-text entailment, etc.) [10, 11]. Some approaches leverage properties of tables together with linearization. For example, [16] first applies a *horizontal transformer* to get

text-row representations for rows in the table, then a *vertical transformer* between all text-row representations to obtain a single text-table representation.

3.2.3 Graph Representation

There is another line of work to represent the table based on graph structure. We can define relations between question tokens and DB entities (e.g. table names, column names, cell values). Example relations include *table-column affiliation* between tables and columns; *primary-foreign key* between columns; *token-table/column mention* between question tokens and tables or columns; etc. Such relations form a graph where each token / DB entity is a node, and each existing relation is an edge. An example of such graph is shown in Figure 2b. Given such a graph, we can use graph neural network to encode it [17, 18]. We can also use the relations directly to enhance linearization-based methods. For example, we can add a *relational bias* term in self-attention layers in transformers. Each relation has a learnable bias vector; when computing the attention weights between two tokens, if they exhibit a relation, we add the corresponding bias vector into the token representation [19, 20] (similar ideas are also studied in table-only representation, such as in [11]).

3.3 Structured Data Representation in Speech-to-SQL

We introduce our research on speech-to-SQL.[1] Most existing studies focus on text-to-SQL which is to parse natural language *text* utterances into executable SQL queries. Motivated by the rise of speech-driven digital assistants on smartphones, tablets, and other small handheld devices, we study the task of parsing spoken natural language (in audio) to executable SQL queries (speech-to-SQL parsing). A speech-to-SQL parser has a number of potential use scenarios for quick and convenient data look-up, such as patient caring (in healthcare domain), business meeting, or driving. In this work, we study ways to improve speech-to-SQL parsers. We will also highlight the representation of structured data, i.e. the DB, in this study.

3.3.1 Baselines

An obvious solution of speech-to-SQL parsing is to first pass the speech audio through an automatic speech recognizer (ASR), and then issue the top-ranked ASR transcription to a text-to-SQL parser which produces the final SQL. We name it the *blackbox* baseline. A drawback of this baseline is that no attempt is made to deal with ASR errors. Figure 3(A) shows an example of such errors. Passing ASR errors to the text-to-SQL parser is unlikely to produce the correct SQL.

For ASR error correction, we import two more baselines from previous work: *re-ranker* and *S2S-rewriter* baseline. The re-ranker takes the top-K candidate transcriptions from ASR and re-rank them to select the best transcription [21]. In contrast, the S2S-rewriter directly rewrites an ASR transcription into a better sentence, as a S2S task [22]. Both methods do not incorporate the DB information.

3.3.2 Method Overview

We propose a two-stage ASR error correction method. First, we *tag* the input sentence (an ASR transcription) to identify tokens that are incorrect and should be edited. Second, we edit the sentence using a *blank-filling* model, where the tagged incorrect parts of the sentence are regarded as blanks and filled by predicted tokens. Moreover, we incorporate the backend DB information into the model by adding the DB schema and relevant cell values into the input sentence, in the above-mentioned *linearized* manner. Figure 3 illustrates the framework of our method.

[1] Under submission. We focus on the high-level ideas and omit the details in order not to violate submission policies.

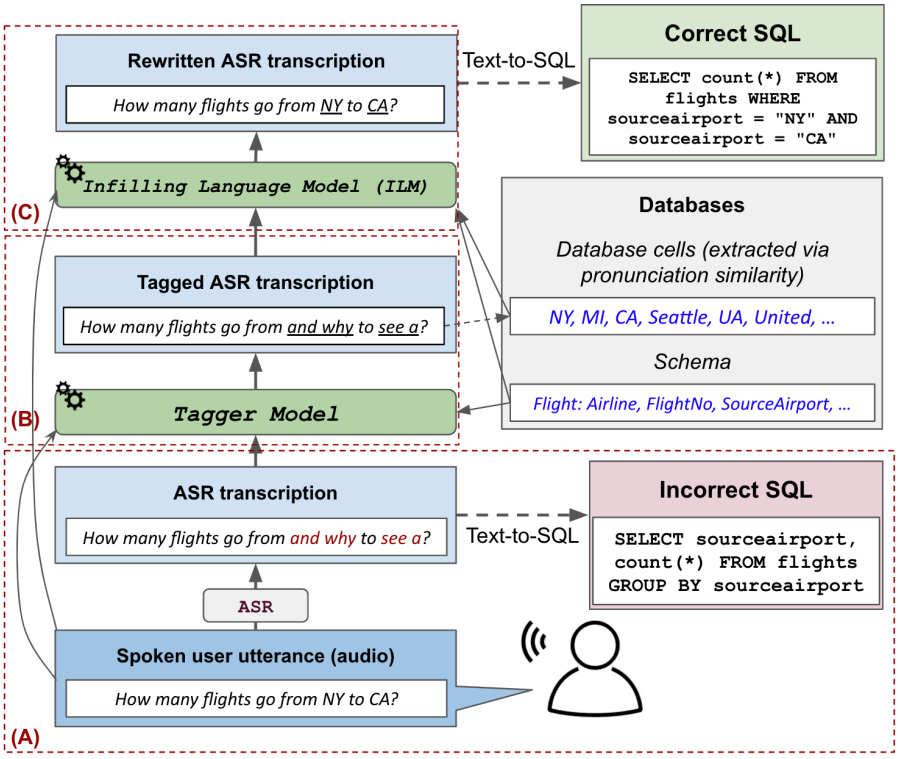


Figure 3: Overview of the ASR correction method for speech-to-SQL. (A) Directly passing the erroneous ASR transcription to text-to-SQL parser will likely produce wrong SQL output. We apply a Tagger (B) and an ILM (blank-filling) rewriter (C) to fix the transcription, incorporating audio features and DB information.

3.3.3 Experiments and Analysis

Datasets Experiments are conducted on the Spider dataset mentioned above. To make it usable for evaluation speech-to-SQL methods, we used Amazon Polly, a text-to-speech (TTS) service, to automatically transform the user queries from text to audio. We used Amazon Transcribe as the raw ASR transcriber. Besides, since our method focuses on ASR correction, we utilize existing models as the backend text-to-SQL parser during evaluation. We use RAT-SQL [19] and T5 (T5-base and T5-large) [13], which are representative and competitive text-to-SQL parsers.

Evaluation Metrics We test baseline methods and our proposed methods on two categories of metrics. The first category is on *text accuracy*, for which we use *word error rate (WER)* and *BLEU score*. The second category measures the final *SQL accuracy*, for which we use *exact match rate* and *execution match rate* of the SQL predictions against ground truth.

Results We test and compare our proposed methods to all baseline methods (blackbox, re-ranker, S2S-rewriter) on Spoken Spider. Results show that our proposed method can outperform all baseline methods on both category of metrics, regardless of backend parser. We also conducted ablation studies to profile the source of performance gain. The results show that the tagging + blank-filling pipeline, compared to re-ranker or S2S-rewriter, is the major source of improvement. Nonetheless, the incorporation of structured data, i.e. DB information, also provides non-negligible benefits. We would also like to highlight that, retrieving and adding relevant cells into the input can substantially improve the performance, compared to only using the DB schema.

4 Task-oriented Dialog Systems

4.1 Real Application Example

As mentioned, a TOD system can help users complete actions more easily, especially in complex scenarios where multi-turn interactions are needed. One of the real use cases is on *data plotting*. Plotting is a very common practice for visualizing data and mathematical functions. Existing plotting libraries such as `matplotlib` support a decent range of functionalities. However, using such libraries can be difficult for novice users and time consuming even for experts, due to both the hardness of programming and complexity of our detailed plotting needs. It thus motivates us to build a TOD system that helps human complete plotting tasks by interacting with users through natural language.

In what follows, we first provide a more general and formal introduction to the TOD systems. We then survey related work in this direction, and introduce our own work on the above-mentioned plotting task.

4.2 Formal Introduction

A TOD system runs in one or several *domains* and requires a domain-specific *ontology*. Intuitively, an ontology is a set of slots and values for each slot, representing the domain-specific information required by the system or the user. Several example ontologies are shown in Figure 4.

act type	inform* / request* / select^{123} / recommend/^{123} / not found^{123} request booking info^{123} / offer booking^{1235} / inform booked^{1235} / decline booking^{1235} welcome* /greet* / bye* / reqmore*
slots	address* / postcode* / phone* / name^{1234} / no of choices^{1235} / area^{123} / pricerange^{123} / type^{123} / internet^{2} / parking^{2} / stars^{2} / open hours^{3} / departure^{45} destination^{45} / leave after^{45} / arrive by^{45} / no of people^{1235} / reference no.^{1235} / trainID^{5} / ticket price^{5} / travel time^{5} / department^{7} / day^{1235} / no of days^{123}

(a) The ontology of MultiWOZ [26]. For each slot, the upperscript indicates the domain indexes it belongs to. *: universal, 1: restaurant, 2: hotel, 3: attraction, 4: taxi, 5: train, 6: hospital, 7: police.

Slot	Requestable	Informable
area	yes	yes. 5 values; north, south, east, west, centre
food	yes	yes, 91 possible values
name	yes	yes, 113 possible values
pricerange	yes	yes, 3 possible values
addr	yes	no
phone	yes	no
postcode	yes	no
signature	yes	no

(b) The ontology of DSTC2 [25]. The domain is restaurant booking.

Figure 4: Sample ontology of existing datasets.

There are two main categories of TOD system design: pipeline-based and end-to-end [27]. A pipeline-based TOD system consists of a collection of separate modules, including natural language understanding (NLU), dialog state tracking (DST), dialog policy, and natural language generation (NLG). In contrast, an end-to-end TOD system directly takes the current user utterance and dialog history as input, and predicts the action or response.

Structured data in TOD systems is a complex topic, because a TOD system involves several different types of structured data, mainly including two categories: *internal* and *external* structured data. External structured data include the backend DB or KG, similar to the structured data for semantic parsing as mentioned above. Internal structured data include *dialog state* and *dialog act*. Dialog state is the output of the dialog state tracking component. It includes the user intents and specified values for each slot. It can be transformed into a DB query to search for relevant information, which may also be used by the model to decide the response. Dialog act is the output of dialog policy component. It controls what the system response wants to achieve. Basically, for system response generation we care about "what to say" and "how to say it"; the dialog act is the "what to say" part.

4.3 Structured Data Representation in Task-oriented Dialog Systems

4.3.1 External

Database (DB) In TOD systems, we utilize the backend DB by queries based on the dialog state (the user-specified slot values are conditions in the query). How the DB query results are utilized depends on the task setting, or practically, the dataset. On certain datasets, the systems are required to use the full content. For example, in the bAbI dialog dataset [23], given all the records satisfying user-specified conditions, the system is expected to "learn" to rank results by the field named "rating". In such scenarios, the system usually regard the records as sentences in the dialog history. On the other hand, many datasets do not have the explicit requirement [25, 26, 28]. As a result, a widely adopted heuristic is to simply use the *number of records* in the query results, instead of the full results content [28–30]. The input and output of the model may be *delexicalized*, i.e. using placeholders for entities (e.g. `[v.HOTEL_NAME]` for a hotel name), which are replaced by actual entities in the query results during post-processing. In this way, in the dialog model, the query results are no longer structured data, but a single token (representing the number) that can be simply represented using look-up embeddings.

Knowledge Graph (KG) Knowledge graphs (KG) are frequently used in the task of *knowledge-grounded dialogs*, in which the user chats with the dialog system to ask questions, learn knowledge or gather information, and the system responses based on external knowledge. This task does not rigorously belong to the definition of TOD, since the system does not make explicit backend API calls. Nonetheless, this task is highly related to TOD in the sense of retrieving relevant information from external data. One way to represent KG is to simply use the (subj, rel, obj) triplet form. For example, in methods based on memory networks [31, 32], each triplet in the KG are vector-embedded and added to the memory bank. During response generation, the token prediction probability includes a term to copy tokens from the KG triplets. Another line of work leverages the graph-structure of KG by "walking" on the edges, simulating human reasoning. These methods can use the encoding vector of current turn information as the initial state. They may iteratively update the state using a gated recurrent cell, each step attending to all walkable nodes [36], or directly predict a sequence of relations to decide the walking path [37, 38].

4.3.2 Internal

As mentioned, internal structured data in TOD systems mainly include dialog state and dialog act. Some fully end-to-end models do not have explicit modeling for internal structured data [31, 32]. However, it decreases the transparency, interpretability and controllability of the model. Thus, despite of the competitive performance of such models, it is still beneficial to explicitly predict and incorporate the internal structured data. Here we put more attention on incorporation and less on prediction. On prediction side, dialog state tracking and dialog policy learning are both popular research topics and involve a large variety of task-specific methods, which are out of the scope here.

Linearization A straightforward yet useful way to represent internal structured data is *linearization*, similar to above-mentioned DB linearization in semantic parsing. We represent the dialog state and/or dialog act as token sequences, and encode them as text. This type of method can be used for both prediction and incorporation, creating a unified pipeline for end-to-end dialog model where the "internal states" are always text. For example, in [33], the dialog model is formulated as a two-stage sequence-to-sequence (S2S) model. First, based on previous dialog state and dialog history, predict current dialog state (this step is essentially dialog state tracking); second, based on dialog history and current dialog state, generate the response. [34] considers TOD as a language modeling (LM) task by concatenating dialog history, dialog state, dialog act and response all together as a token sequence. They fine-tune a pretrained LM, GPT2, to predict dialog state, dialog act and response based on the input dialog history.

Other Specific Methods There are methods exploiting the determined structures of such internal structured data in order to effectively model them. For example, [28] assumes a fixed domain and ontology, thus dialog state can be treated simply as a probability distribution per slot. [35] exploits the (intent, slot, value) triplet structure of dialog act to build 3-layer tree to predict and control the encoder of dialog act. Such methods can potentially have simpler design or have better performances on specific datasets; however, it is harder for them to generalize to other datasets or domains.

4.4 Structured Data Representation in a Plotting Agent

We introduce our original work on the novel task of *plotting agent*, by which we refer to the above-mentioned TOD system for data plotting [39]. Notice that this plotting agent is targeted at manipulating the plot appearance (colors, shapes, sizes, etc.) instead of the underlying data. In this work, we collected a large-scale dataset for training a plotting agent, and conducted experiments on competitive methods to compare and analyse their performances. We also showcase the influence of structured data representations on model performances.

4.4.1 Problem Definition

We aim to develop a *conversational plotting agent* that takes natural language instructions and updates the plot accordingly. The agent is designed conversational because plots can be complex, making it difficult to describe everything at once; thus, users may want to tune the appearance of their plot through multiple turns. Technically, a conversational plotting agent is framed as a TOD system. It has only one domain, which is plot control. We manually defined the domain ontology to include several plot types and a large number of slots, based on `matplotlib` documentation and the common needs based on our experiences. Figure 5 illustrates example slots for some of the plot types. The full ontology is shown in Table 2. Different plot types have different sets of slots, yet some slots are shared across plot types. For example, the slot "X-axis scale" is relevant to the X-axis, thus it is applicable in any plot type with an X-axis, including line chart, bar plot, contour plot, etc. For modeling purposes, the plot type can also be seen as a slot in the ontology. Notice that, in the current definition, a conversational plotting agent is simpler than a "full" TOD system, because the system directly uses dialog states as responses and does not have to generate text responses.[2]

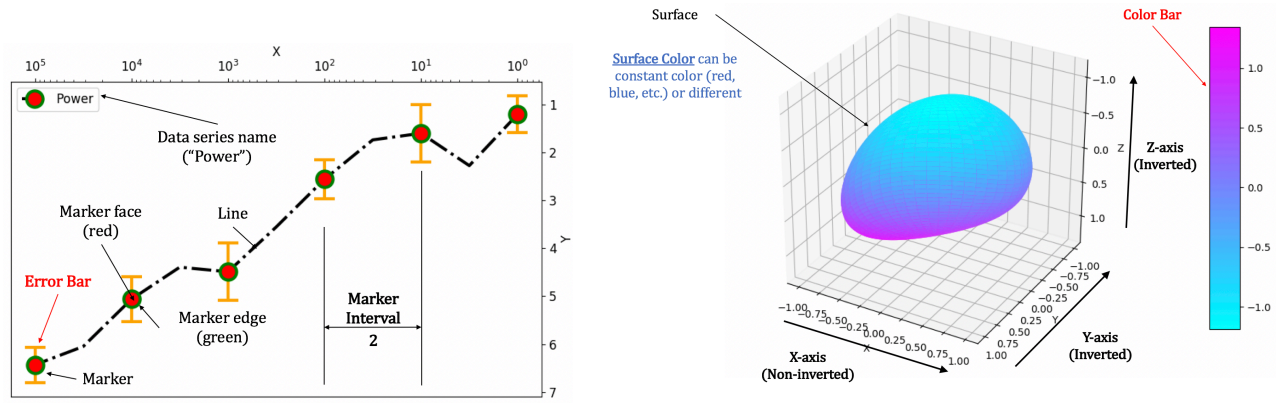
4.4.2 Dataset Overview

To enable training neural models for the new task, we collected a dataset, *ChartDialogs*, which consists of actual human dialogs on completing plotting tasks. We used Amazon Mechanical Turk for dataset collection. For each dialog, we engage two workers simultaneously to play the *user* and *system* respectively, and randomly generate a *target plot*. The target plot is only visible to the user worker. It stands for the plot that an actual user wants in real world scenarios. During the dialog, the user worker has to describe the plot to the system worker to accurately reproduce the exact plot to complete the task.

4.4.3 Methods

We assessed the performance of various methods for training TOD system on our ChartDialogs dataset. We experimented with two categories of methods: S2S-based and classification-based. For S2S-based methods, we linearize the dialog state in similar manner with previous work [33]. The model takes as input the user utterance and linearized dialog state, and predicts a linearized *dialog state update*, which includes the slot values that should be updated. We use LSTM as both encoder and decoder architecture for the S2S model. For classification-based methods, we exploit the specific design of our domain ontology that the possible values of

[2]We wrote a simple script that can take as input the dialog state (plot type and other slot values), to generate the actual plot image using `matplotlib` and display it to the user. Thus, plot controlling is equivalent to dialog state tracking.



(a) Line Chart

(b) 3D surface

Figure 5: Illustration of two of ChartDialog plot types. **(a) Line Chart** has slots such as *Line Style, Marker Interval*. **(b)** A **3D Surface** has slots such as *Surface Color*.

each slot come from a finite value pool. Thus, we can treat dialog state tracking as a classification problem for each slot separately. The input format is the same as above, i.e. the concatenation of user utterance and linearized dialog state. We experimented using bag-of-word, LSTM and BERT [47] as input encoding methods, and train a logistic-regression or multi-layer perceptron (MLP) classifier heads[3] for each slot on top of the encoded input representation. In detail, we try three different classification methods: (i) **MaxEnt**, using bag-of-word embedding and logistic regression classifier; (ii) **LSTM+MLP**, using LSTM as input encoder and MLP classifier heads; and (iii) **BERT+MLP**, using BERT (parameters frozen) as input encoder and MLP classifier heads.

As a relevant detail, we experimented using different *granularity* for dialog state linearization, SPLIT, SINGLE and PAIR, detailed in Table 3. Conceptually, SPLIT best leverages the semantic overlap between different slots and values, while PAIR is the most succinct with regard to sequence length. In the result section, we shed light on how this design choice influences the model performances.

4.4.4 Experiments

Evaluation Metrics We evaluate the model performance on a turn-based manner, i.e. use each dialog turn as a data point and compare the model prediction with ground truth (human worker choices). For evaluation metrics, we use *exact match rate* which is the proportion of dialog state predictions fully matching the ground truth; and *Slot-F1* which measures the performance on slot-level.[4]

Results The main results are shown in Table 4. The S2S method largely outperforms classification-based methods, despite the fact that classification-based methods exploit the domain-specific design. Our hypothesis is that, the sequence predictor (decoder) in S2S model learns implicit inter-dependencies between slots (e.g. certain slots are only active for certain plot types), while the separated classifiers do not. Comparing classification methods, the simplest method, MaxEnt (with PAIR granularity) is competitive and outperforms baseline neural

[3] An MLP consists of several fully-connected layers with non-linear activation functions, finally applying a softmax layer to predict the probabilities of each class.

[4] We use F1-score instead of accuracy because in most cases, most slots are inactive and have the *[None]* value. That causes uneven distributions among slot values, thus we use F1-score.

	Plot Types	Slots
1.	Axes	Polarize, X-axis Scale, Y-axis Scale, X-axis Position, Y-axis Position, Invert X-axis, Invert Y-axis, Grid Line Type, Grid Line Style,Grid Line Width, Grid Line Color, Font Size
2.	3D Surface	Color map, Invert X-axis, Invert Y-Axis, Invert Z-Axis
3.	Bar Chart	Bar Orientation, Bar Height, Bar Face Color, Bar Edge Width, Bar Edge Color, Show Error Bar, Error Bar Color, Error Bar Cap Size, Error Bar, Cap Thickness, Data Series Name
4.	Contour/Filled	Contour Plot Type, Number of levels, Color Map, Color Bar Orientation, Color Bar Length, Color Bar Thickness
5.	Contour/Lined	Contour Plot Type, Lined Style, Line Width
6.	Histogram	Number of Bins, Bar Relative Width, Bar Face Color, Bar Edge Width, Bar Edge Color, Data Series Name
7.	Matrix	Color Map, Invert X-axis, Invert Y-axis
8.	Line Chart	Line Style, Line Width, Line Color, Marker Type, Marker Size, Marker Face Color, Marker Edge Color, Marker Interval, Data Series Name, Show Error Bar, Error Bar Color, Error Bar Cap Size, Error Bar Cap Thickness
9.	Pie Chart	Exploding Effect, Precision Digits, Percentage tags' distance from center, Label tag's distance from center, Radius, Section Edge Width, Section Edge Color
10.	Polar	Polarize, Grid Line Type, Grid Line Style, Grid Line Width, Grid Line Color, Font Size
11.	Scatter	Polarize, Marker Type, Marker Size, Marker Face Color, Marker Edge Width, Marker Edge Color, Color Map, Color Bar Orientation, Color Bar Length Color Bar Thickness
12.	Streamline	Density, Line Width, Line Color, Color Map, Arrow Size, Arrow Style

Table 2: Plot types and slots in our dataset

Granularity	Description	Example
(Dialog state)	The dialog state to linearize	`(plot_type = line chart, line_color = blue, marker_type = circle)`
PAIR	Combined slot-value pair as a token	"plot_type:line_chart line_color:blue marker_type:circle"
SINGLE	Each slot or value as a single token	"plot_type line_chart line_color blue marker_type circle"
SPLIT	Split slot or values into natural language tokens	"plot type : line chart | line color : blue | marker type : circle"

Table 3: Explanation of dialog state linearization granularity.

models. The unsatisfactory performance is possibly because their non-robustness or overfitting to noises in the dataset.

Based on the best-performing S2S method, we also conducted ablation study to verify the usefulness of user utterance and dialog state in the input. User utterances are useful as expected, as without user utterances the prediction would be an educated guess. The dialog states are also useful, which is a positive sign for the importance of (internal) structured data in TOD. Intuitively, the benefits of dialog state could be to provide a better semantic context, i.e. more relevant information, for the model. Comparing the linearization granularity for the S2S model, the SINGLE setting performs the best. It possibly implies that, among the three granularity settings, SINGLE achieves a proper trade-off between capturing slot / value semantics and controlling the sentence length.

5 Conclusion

In this paper, we reviewed relevant tasks of natural language interfaces, semantic parsing (focusing on text-to-SQL) and task-oriented dialog (TOD) systems. We survey representative methods on these tasks, especially focusing on the perspective of representing and incorporating structured data, both external and internal. In general, linearization methods are widely adopted. They are applicable for almost all types of structured data

[5]Due to the word-piece tokenization used in BERT, the SINGLE and PAIR linearizations are also tokenized to granularity level of SPLIT, therefore we do not report their performance.

Methods	Exact Match			Slot F1		
	SPLIT	SINGLE	PAIR	SPLIT	SINGLE	PAIR
S2S	0.601	**0.613**	0.591	0.874	**0.893**	0.885
S2S-NoState	0.525	0.549	0.535	0.847	0.866	0.863
S2S-NoUtterance	0.060	0.047	0.046	0.316	0.306	0.155
MaxEnt	0.196	0.265	***0.422***	0.677	0.734	***0.806***
LSTM+MLP	0.328	0.324	0.325	0.714	0.712	0.724
BERT+MLP	0.311	n/a[5]	n/a[5]	0.723	n/a[5]	n/a[5]

Table 4: Exact match plotting performance.

while exhibiting competitive performances. Nonetheless, for different types of structured data, there are still opportunities for further performance gains by designing methods that capitalize on their distinct properties. We also present our original studies on the relevant tasks, and discuss our findings regarding the topic of structured data representation.

For future work on NLIs and structured data representation, many opportunities and challenges are still ripe for exploration. Despite the increasing performance on benchmarks, neural-based NLIs are still not widely deployed in the real world. Identifying and analyzing the challenges that remain when porting to new domains, tasks, datasets or real-world scenarios is one line of future work. Another future direction is a systematic study to find similarities among a subset of different tasks, or even all NLI-related tasks, to develop methods that are generalizable to new tasks with similar properties. Furthermore, obtaining data for NLIs in new domains, even a small amount of data for few-shot or fine-tuning remains a time-consuming endeavor that requires complex crowd-sourcing pipelines, and can be expensive. Coming up with ways to quickly obtain new data for training NLIs in new domains, and tasks is another direction for future work.

References

[1] Winograd, Terry. "Procedures As A Representation For Data In A Computer Program For Understanding Natural Language." (1971).

[2] Karamcheti, Siddharth, Dorsa Sadigh and Percy Liang. "Learning Adaptive Language Interfaces through Decomposition." ArXiv abs/2010.05190 (2020): n. pag.

[3] Zelle, John M. and Raymond J. Mooney. "Learning to Parse Database Queries Using Inductive Logic Programming." AAAI/IAAI, Vol. 2 (1996).

[4] Pasupat, Panupong and Percy Liang. "Compositional Semantic Parsing on Semi-Structured Tables." ArXiv abs/1508.00305 (2015): n. pag.

[5] Artzi, Yoav and Luke Zettlemoyer. "UW SPF: The University of Washington Semantic Parsing Framework." ArXiv abs/1311.3011 (2013): n. pag.

[6] Dong, Li and Mirella Lapata. "Language to Logical Form with Neural Attention." ArXiv abs/1601.01280 (2016): n. pag.

[7] Dong, Li and Mirella Lapata. "Coarse-to-Fine Decoding for Neural Semantic Parsing." ACL (2018).

[8] Cheng, Jianpeng, Siva Reddy, Vijay A. Saraswat and Mirella Lapata. "Learning an Executable Neural Semantic Parser." Computational Linguistics 45 (2019): 59-94.

[9] Yu, Tao, Rui Zhang, Kai-Chou Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Z Li, Qingning Yao, Shanelle Roman, Zilin Zhang and Dragomir R. Radev. "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task." EMNLP (2018).

[10] Herzig, Jonathan, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno and Julian Martin Eisenschlos. "TaPas: Weakly Supervised Table Parsing via Pre-training." ArXiv abs/2004.02349 (2020): n. pag.

[11] Yang, Jingfeng, Aditya Gupta, Shyam Upadhyay, Luheng He, Rahul Goel and Shachi Paul. "TableFormer: Robust Transformer Modeling for Table-Text Encoding." ArXiv abs/2203.00274 (2022): n. pag.

[12] Scholak, Torsten, Nathan Schucher and Dzmitry Bahdanau. "PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models." ArXiv abs/2109.05093 (2021): n. pag.

[13] Xie, Tianbao, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, Victor Zhong, Bailin Wang, Chengzu Li, Connor Boyle, Ansong Ni, Ziyu Yao, Dragomir Radev, Caiming Xiong, Lingpeng Kong, Rui Zhang, Noah A. Smith, Luke Zettlemoyer and Tao Yu. "UnifiedSKG: Unifying and Multi-Tasking Structured Knowledge Grounding with Text-to-Text Language Models." ArXiv abs/2201.05966 (2022): n. pag.

[14] Lin, Xi Victoria, Richard Socher and Caiming Xiong. "Bridging Textual and Tabular Data for Cross-Domain Text-to-SQL Semantic Parsing." EMNLP Findings (2020).

[15] Yu, Tao, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Richard Socher and Caiming Xiong. "GraPPa: Grammar-Augmented Pre-Training for Table Semantic Parsing." ArXiv abs/2009.13845 (2021): n. pag.

[16] Yin, Pengcheng, Graham Neubig, Wen-tau Yih and Sebastian Riedel. "TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data." ArXiv abs/2005.08314 (2020): n. pag.

[17] Bogin, Ben, Matt Gardner and Jonathan Berant. "Representing Schema Structure with Graph Neural Networks for Text-to-SQL Parsing." ArXiv abs/1905.06241 (2019): n. pag.

[18] Cao, Ruisheng, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu and Kai Yu. "LGESQL: Line Graph Enhanced Text-to-SQL Model with Mixed Local and Non-Local Relations." ACL (2021).

[19] Wang, Bailin, Richard Shin, Xiaodong Liu, Oleksandr Polozov and Matthew Richardson. "RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers." ACL (2020).

[20] Hui, Binyuan, Ruiying Geng, Lihan Wang, Bowen Qin, Bowen Li, Jian Sun and Yongbin Li. "S2SQL: Injecting Syntax to Question-Schema Interaction Graph Encoder for Text-to-SQL Parsers." ArXiv abs/2203.06958 (2022): n. pag.

[21] Weng, Yue, Sai Sumanth Miryala, Chandra Khatri, Runze Wang, Huaixiu Zheng, Piero Molino, Mahdi Namazifar, Alexandros Papangelis, Hugh Williams, Franziska Bell and Gokhan Tur. "Joint Contextual Modeling for ASR Correction and Language Understanding." ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (2020): 6349-6353.

[22] Mani, Anirudh, Shruti Palaskar, Nimshi Venkat Meripo, Sandeep Konam and Florian Metze. "ASR Error Correction and Domain Adaptation Using Machine Translation." ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (2020): 6344-6348.

[23] Bordes, Antoine and Jason Weston. "Learning End-to-End Goal-Oriented Dialog." ArXiv abs/1605.07683 (2017): n. pag.

[24] Wei, Wei, Quoc V. Le, Andrew M. Dai and Jia Li. "AirDialogue: An Environment for Goal-Oriented Dialogue Research." EMNLP (2018).

[25] Henderson, Matthew, Blaise Thomson and J. Williams. "The Second Dialog State Tracking Challenge." SIGDIAL Conference (2014).

[26] Budzianowski, Paweł, Tsung-Hsien Wen, Bo-Hsiang Tseng, Iñigo Casanueva, Stefan Ultes, Osman Ramadan and Milica Gasic. "MultiWOZ - A Large-Scale Multi-Domain Wizard-of-Oz Dataset for Task-Oriented Dialogue Modelling." EMNLP (2018).

[27] Zhang, Zheng, Ryuichi Takanobu, Minlie Huang and Xiaoyan Zhu. "Recent Advances and Challenges in Task-oriented Dialog System." ArXiv abs/2003.07490 (2020): n. pag.

[28] Rojas-Barahona, Lina Maria, Milica Gašić, Nikola Mrksic, Pei-hao Su, Stefan Ultes, Tsung-Hsien Wen, Steve J. Young and David Vandyke. “A Network-based End-to-End Trainable Task-oriented Dialogue System.” EACL (2017).

[29] Liu, Bing, Gökhan Tür, Dilek Z. Hakkani-Tür, Pararth Shah and Larry Heck. “Dialogue Learning with Human Teaching and Feedback in End-to-End Trainable Task-Oriented Dialogue Systems.” NAACL (2018).

[30] Zhang, Yichi, Zhijian Ou, Huixin Wang and Junlan Feng. “A Probabilistic End-To-End Task-Oriented Dialog Model with Latent Belief States towards Semi-Supervised Learning.” ArXiv abs/2009.08115 (2020): n. pag.

[31] Madotto, Andrea, Chien-Sheng Wu and Pascale Fung. “Mem2Seq: Effectively Incorporating Knowledge Bases into End-to-End Task-Oriented Dialog Systems.” ACL (2018).

[32] Chen, Xiuyi, Jiaming Xu and Bo Xu. “A Working Memory Model for Task-oriented Dialog Response Generation.” ACL (2019).

[33] Lei, Wenqiang, Xisen Jin, Min-Yen Kan, Zhaochun Ren, Xiangnan He and Dawei Yin. “Sequicity: Simplifying Task-oriented Dialogue Systems with Single Sequence-to-Sequence Architectures.” ACL (2018).

[34] Ham, Dong-hyun, Jeong-Gwan Lee, Youngsoo Jang and Kyungmin Kim. “End-to-End Neural Pipeline for Goal-Oriented Dialogue Systems using GPT-2.” ACL (2020).

[35] Chen, Wenhu, Jianshu Chen, Pengda Qin, Xifeng Yan and William Yang Wang. “Semantically Conditioned Dialog Response Generation via Hierarchical Disentangled Self-Attention.” ACL (2019).

[36] Moon, Seungwhan, Pararth Shah, Anuj Kumar and Rajen Subba. “OpenDialKG: Explainable Conversational Reasoning with Attention-based Walks over Knowledge Graphs.” ACL (2019).

[37] Cohen, William W., Haitian Sun, R. Alex Hofer and Matthew A. Siegler. “Scalable Neural Methods for Reasoning With a Symbolic Knowledge Base.” ArXiv abs/2002.06115 (2020): n. pag.

[38] Tuan, Yi-Lin, Sajjad Beygi, Maryam Fazel-Zarandi, Qiaozi Gao, Alessandra Cervone and William Yang Wang. “Towards Large-Scale Interpretable Knowledge Graph Reasoning for Dialogue Systems.” ArXiv abs/2203.10610 (2022): n. pag.

[39] Shao, Yutong and Ndapa Nakashole. “ChartDialogs: Plotting from Natural Language Instructions.” ACL (2020).

[40] M. Luong, E. Brevdo, and R. Zhao. Neural machine translation (seq2seq) tutorial. https://github.com/tensorflow/nmt, 2017.

[41] Sutskever, Ilya, Oriol Vinyals and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks.” NIPS (2014).

[42] Hochreiter, Sepp and Jürgen Schmidhuber. “Long Short-Term Memory.” Neural Computation 9 (1997): 1735-1780.

[43] Cho, Kyunghyun, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk and Yoshua Bengio. “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation.” EMNLP (2014).

[44] Bahdanau, Dzmitry, Kyunghyun Cho and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate.” CoRR abs/1409.0473 (2015): n. pag.

[45] Luong, Thang, Hieu Pham and Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation.” EMNLP (2015).

[46] Vaswani, Ashish, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin. “Attention is All you Need.” ArXiv abs/1706.03762 (2017): n. pag.

[47] Devlin, Jacob, Ming-Wei Chang, Kenton Lee and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” ArXiv abs/1810.04805 (2019): n. pag.

[48] Radford, Alec, Jeff Wu, Rewon Child, David Luan, Dario Amodei and Ilya Sutskever. “Language Models are Unsupervised Multitask Learners.” (2019).

Quill: A Declarative Approach for Accelerating Augmented Reality Application Development

Codi Burley, Ritesh Sarkhel, and Arnab Nandi
The Ohio State University
{burley.66, sarkhel.5, nandi.9}@osu.edu

Abstract

Data we encounter in the real-world such as printed menus, business documents, and nutrition labels, are often ad-hoc. Valuable insights can be gathered from this data when combined with additional information. Recent advances in computer vision and augmented reality have made it possible to understand and enrich such data. Joining real-world data with remote data stores and surfacing those enhanced results in place, within an augmented reality interface can lead to better and more informed decision-making capabilities. However, building end-user applications that perform these joins with minimal human effort is not straightforward. It requires a diverse set of expertise, including machine learning, database systems, computer vision, and data visualization. To address this complexity, we present Quill *– a framework to develop end-to-end applications that model augmented reality applications as a join between real-world data and remote data stores. Using an intuitive domain-specific language, Quill accelerates the development of end-user applications that join real-world data with remote data stores. Through experiments on applications from multiple different domains, we show that Quill not only expedites the process of development, but also allows developers to build applications that are more performant than those built using standard developer tools, thanks to the ability to optimize declarative specifications. We also perform a user-focused study to investigate how easy (or difficult) it is to use Quill for developing augmented reality applications than other existing tools. Our results show that Quill allows developers to build and deploy applications with a lower technical background than building the same application using existing developer tools.*

1 Introduction

We gather information in our everyday life not only through digital interfaces such as mobile devices, computers, and wearables, but also through multiple physical media that surround us in the real world, such as movie posters, billboards, chalkboard menus, and grocery lists. Real-world data is diverse and rich in its complexity. It is also *one-size-fits-all* – unlike digital data which is often personalized or transformed based on the downstream task, we all see the same real-world data irrespective of our information needs. This makes it challenging to devise a generalizable solution to gather insights from real-world data. For example, if *Alice* wants to find out "recent reviews of the most popular dish in a newly opened restaurant", she will need to formulate her information need as a tangible query first, look up the results of that query in a browser-based search engine, and then filter down

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

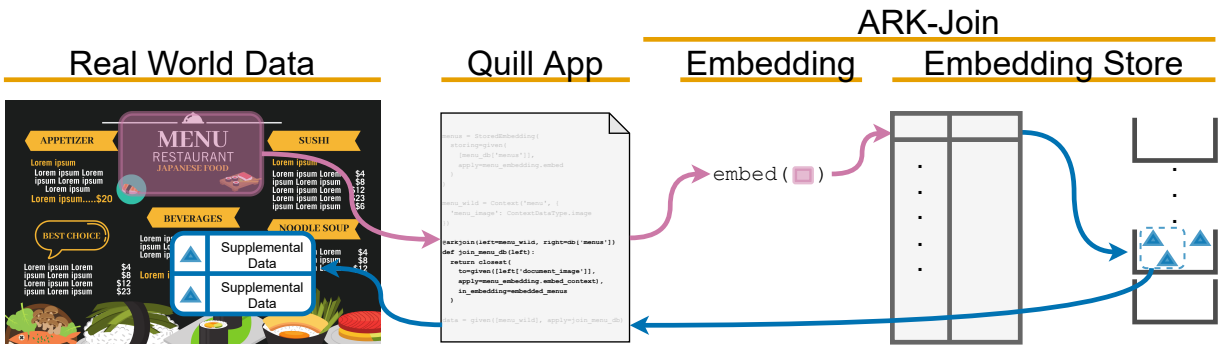


Figure 1: This figure demonstrates the flow of an application developed using the *Quill* framework, performing an Augmented Reality-adapted Keyless Join (ARK-Join). An AR-enabled client sends requests in the form of data elements from a *real-world context* to the Quill app, which is defined using a script written in a domain-specific language, called *QWL*. The ARK-Join operation then finds sufficiently similar data elements from a specified *remote source* by searching over an indexed *embedding store*. Results of this join operation are sent to the client which uses this data to augment the user's view with supplementary information.

the returned results to find an appropriate answer. If *Bob* wants to find out the reviews of another dish from the same restaurant, he would have to go through the same steps again. Real-world data is also *incomplete*: it may provide little to no context with respect to the information a consumer may need. For example, restaurant menus often lack supplementary information such as a list of ingredients and allergens present in a dish. As a result, if *Alice* wants to discern if a dish in her favorite restaurant has nuts in it, she has to manually perform what is essentially a join between real-world data, i.e., the name or image of the dish, and a remote data store, i.e., a nutritional database that contains the list of ingredients.

Augmented reality (AR), a technology to overlay the physical world with digital information, provides a useful medium for enriching real-world data with supplementary information. The rapid advancement of cameras and computational power in consumer-grade mobile devices [1] have made AR widely available in recent years. In prior work [2] titled "ARQuery", we have shown that AR can be an effective way to explore real-world data interactively. For example, interactive querying of each dish in the menu through an AR-enabled interface for relevant supplementary information (e.g., ingredients used, customer reviews) using simple gesture-based interactions [3] can be vastly more efficient than manually looking up a browser-based search engine. However, as depicted in the example below, building a robust capability towards joining real-world data through an interactive, AR-enabled view without handcrafted features and minimal human supervision requires extensive machine learning capabilities [4]. To address these needs, we provide a domain-specific language (DSL) for defining applications that join real-world data with semantically similar data elements in remote data stores in an easy-to-use framework, called *Quill*.

With Quill, we provide a framework for developing augmented reality applications that enrich the real-world data elements with supplementary information from remote data stores. This framework is centered around an easy-to-use DSL, called ***Quill Workflow Language*** or *QWL*. Using QWL, developers can define various task-specific parameters such as the remote data stores to be joined against, data embedding techniques to be used, and more. We describe all such parameters currently supported by Quill in Section III. Quill's framework then produces a ready-to-deploy application that serves input requests in the form of ad-hoc real-world data. It performs this join operation with minimal human supervision and handcrafted features in its end-to-end workflow. This is achieved by comparing remote data elements to incoming real-world data elements in a shared vector space. The DSL also enables us to define various application-specific parameters, e.g., whether to cache previously computed results locally or to recompute them when comparing against a remote data element. We discuss the challenges addressed by our framework in greater detail after describing a scenario that showcases some of these

challenges and demonstrates how Quill accelerates the development workflow.

Example 1.1: *Alice* is visiting a French cafe. The cafe has a printed menu. *Alice* wants to order an appetizer but wants to know its calorific and allergen information first. She has to pore through the appetizer section of the menu and manually look up the necessary information for each item one at a time. With an application developed using Quill, *Alice* simply views the menu through her phone's live camera view and selects the dish she wishes to know more about in the menu using simple gesture-based interactions. This initiates a join of the real-world data retrieved from the menu, i.e., the dish *Alice* is interested in against a nutritional database. Results of this operation are retrieved from the server running the Quill app and rendered as interactive components in *Alice's* live camera view. Quill enables a developer to create such applications by using a script written in QWL, Quill's DSL. We refer to these applications as Quill apps in the rest of this document. Figure 1 demonstrates the workflow of a typical Quill app. Next, we describe the challenges of developing a framework that enriches real-world data with remote data stores in an augmented reality setting.

Challenge 1: Real-world data is multimodal. Consider the scenario in Example 1.1. Previous works [5–7] suggest that extracting information such as the name of a dish and its price from a live camera view requires considering both the textual and visual properties of the menu. Therefore, a development framework that enables the enrichment of real-world data captured through camera-enabled AR interfaces will need to account for multiple data modalities.

Challenge 2: Modern data stores are often heterogeneous. Instances that contain structured, unstructured, and semi-structured data elements altogether have become a common occurrence. Recent works [8–10] have shown that similarity-based joins such as keyless joins are useful alternatives to traditional join operators for modern data stores due to their heterogeneous nature. However, previous keyless joins have been restricted to singular domains that do not generalize to all kinds of real-world data. Quill enables joining real-world data with remote data stores by implementing a *AR-adapted keyless join* (ARK-join) operation.

Challenge 3: A development framework that enriches real-world data with a remote data store needs to be both *flexible* and *modular*. It should be flexible enough to allow its end-users to define the underlying join operation in a fine-grained way. For example, for the managers of the restaurants *Alice* visits, enhancing a restaurant menu with supplementary information from the inventory database is more useful to plan ahead and keep the pantry stocked. Integration of machine learning components into interactive data systems is an active area of research. Therefore, a development framework should also be modular such that new advances in related areas, e.g., keyless joins, data transformation/embedding functions, and indexing techniques, can be seamlessly integrated.

Challenge 4: The framework should be both *easy-to-use* and *performant*. In other words, compared to applications built without Quill, Quill apps should require less development effort without sacrificing accuracy or latency.

1.1 Technical Contributions

Keeping the above mandates in mind, we describe Quill's main contributions below.

- an augmented reality adapted keyless join (ARK-join) operation to enrich ad-hoc real-world data with heterogeneous data elements from remote data stores,
- an easy-to-use domain-specific language called QWL that allows users to express task specifications in a fine-grained way (including details of the architecture where the application will be deployed),
- a performant framework for developing performant applications that enhance real-world data in a fast and accurate way.

Quill makes it easier to gather insights from real-world data by accelerating the development workflow of applications that enrich real-world data elements with semantically similar data from remote data stores. By using a keyless join, Quill makes it possible to enrich real-world data in a context-aware way with minimal human intervention and handcrafted feature engineering. Through exhaustive experiments on four real-world applications (see Section 6), we show that developing applications is faster using Quill than other developer tools. Moreover, in Section 6.3 we outline how Quill maintains perfomance while remaining usable. We ensure that Quill apps can operate on large remote data stores by indexing data elements in the remote data store to enable faster approximations with tunable accuracy. Our experiments show that Quill apps are more accurate and faster in executing the underlying task than similar applications developed using other developer tools. Our study with a developer audience demonstrates that (a) Quill's DSL is easy to learn, and (b) using Quill led to more succinct codebases while maintaining expressibility and end-to-end performance. Integration of machine learning components in interactive data systems is an active area of research. Fine-grained task specification capabilities afforded by Quill's expressive DSL make it easier for future advances in related areas, e.g., keyless join and embedding techniques, to be seamlessly integrated into Quill's development workflow. We demonstrate this by developing a Quill app that executes a search task over a benchmark text dataset using learned embedding functions from Ember, a recent work by Suri, Ilyas, Ré and Rekastinas [8] (see Section 6.2).

2 Problem Formulation & Definitions

The Quill framework produces a ready-to-deploy application that responds to data enrichment requests by executing an augmented reality-adapted keyless join (ARK-join) specified using a DSL. We provide a formal definition of the ARK-Join operation in Section 2.1. Quill's DSL, called QWL, is capable of defining applications that perform both simple joins, i.e., joining a real-world data element with a remote cloud store, and complex multi-joins (see Fig. 2), i.e., joins between real-world data-elements, a local cache, as well as a remote data store. Before formalizing the problem Quill solves, we define some of its key concepts and related technology first.

2.1 Background and Definitions

Real-world data: All data elements captured in the live camera view of an AR-enabled interface are considered *real-world data*. Real-world data is inherently multimodal.

Remotely stored data: *Remotely stored data* represent those data elements that are stored in cloud-based data stores separately from the real-world data captured in the live camera view of an AR-enabled interface. For example, in the edge-based architecture shown in Fig. 2, the data stored in the cache on edge as well as the data stored in the cloud are considered remotely stored data. Quill enriches real-world data with remotely stored data using an *ARK-Join operation*.

Keyless Join: Quill builds on existing work on keyless join and fuzzy join. A fuzzy join is a commonly used join operator [10–14] in structured data settings used to identify matching records. It matches two records r and s from two different databases R and S if $sim(s, r) > \theta$, where sim represents a similarity function [15] and θ represents a similarity threshold. Among many options, Jaccard similarity, Hamming distance, and Cosine similarity are some of the most common similarity functions used by contemporary researchers. While some recent works [14] have tried to propose scalable fuzzy join operators, they have largely been domain-specific. Consequently, they do not generalize well for data stores where data elements are heterogeneous. At the same time, a number of recent works have proposed a keyless join operator [8–10]. It is an extension of the fuzzy join operator that matches two data-elements r, s if $sim(f_{(s)}(s), f_{(r)}(r)) > \theta$, where $f_{(s)}$ and $f_{(r)}$ are transformations of the data-elements r, s to a shared vector-space. The transformation function f is typically learned from

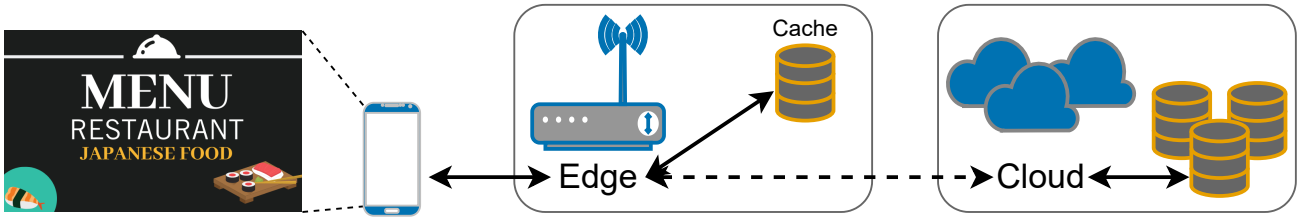


Figure 2: A simplified edge-based architecture with an edge cache and a backing cloud store. Quill can produce applications that fit into various architectures such as this one with a single script written in our DSL.

previous examples. AutoFuzzyJoin [10] searches a configuration space of all possible similarity functions (sim), input transformations (f), and thresholds (θ) to find the optimal configuration. Ember [8] and Termite [9] use machine learning techniques to learn the transformation function f, they use cosine similarity as the similarity function. Keyless join operators are more flexible than traditional fuzzy join operators in capturing semantic information of a data element in a heterogeneous data store.

ARK-Join: To enrich real-world data with remote data stores, Quill employs a keyless join operator, called the augmented reality adapted keyless join or ARK-Join. Summarily, it is a cross-modal keyless join operation extended for augmented reality enabled interfaces. In terms of its basic functionality, ARK-Join is similar to keyless join [8, 9] except it parameterizes the embedding transformation function f as an additional input to the join operator. This allows a developer using the Quill framework the flexibility to define unique transformation functions for data elements from different modalities. In other words, ARK-Join extends the notion of keyless joins beyond data modalities and makes the join operation more generalizable. ARK-Join operator also differs from other existing works on keyless joins such as Ember [8] and Termite [9] in its implementation. It takes advantage of AR-specific constraints, such as the small number of objects that users can track simultaneously at the same time, to optimize the ARK-Join operator for data from camera-enabled AR interfaces. We formalize the ARK-Join operator and describe it in greater detail in Section 3-B.

Quill Apps: Quill exposes a development framework for applications that enriches real-world data with remote data stores through QWL, an intuitive DSL based on an easy-to-learn grammar (see Section 3). We refer to an application produced by Quill's development framework as a *Quill app*. A Quill app is a server application that receives requests to enrich real-world data with semantically similar data elements from remote data stores and responds to these requests by executing an ARK-Join operation (described above) on the remote data store. To develop a Quill app, a developer defines the task specifications and application functionalities using a QWL script. This includes: (a) real-world data provided by the user along with its modality, (b) remote data stores along with methods (e.g. API) for accessing that data, and (c) parameters of the ARK-Join operation which includes the input transformation functions, similarity functions, and similarity threshold. Using the information provided in the QWL script, the Quill framework produces a Quill app. Based on the composition of ARK-Joins that operate on different data sources, such as local caches and remote cloud stores, Quill can seamlessly produce applications with varying architectures such as the one shown in Fig. 2.

2.2 Problem Formulation

Using the concepts introduced above, we now define the problem that Quill solves. Quill provides a development framework for ready-to-deploy applications that execute ARK-Join operations between real-world and remote data elements in an efficient and effective way. Fig. 4 shows the DSL script used to define an application that performs an ARK-join to complete the task described in example 1.1. Quill's development framework is centered

around an easy-to-learn DSL, called QWL. It is based on a grammar that defines the relation between different components of the application that executes the underlying ARK-join operation. We describe this grammar in Section 3.

3 A Grammar for Joining with the Real-World

To ensure that our development framework can express a range of different applications, we base our DSL on a grammar that captures the inner workings of all the components that are inherent in an application. We recognize that while general-purpose systems can be designed to perform well for a variety of different applications, tailoring to specific use cases is challenging [16]. By developing a DSL that is built on an easy-to-learn grammar, we enable the development of such instance-optimized applications for a range of real-world applications. We describe the grammar elements used by Quill's DSL next. They are of three major types: *Data Sources*, *Transformation Functions*, and *ARK-Join Operator*. We finish the section by describing how QWL relates different elements of the grammar using a directed acyclic graph (DAG).

3.1 Data Sources

In Quill, data sources are collections of data elements of varying modalities that originate from one of the data sources defined by our grammar. Data sources can be both real-world data and remotely stored data as defined in Section 2.1. We define three different data sources: *Remote Sources*, *Embedding Stores*, and *Real-World Context*.

Remote Sources: Remote Sources correspond with remotely stored data (defined in Section 2.1). Quill apps execute a join operation on data elements from remote sources against real-world data. Formally, a remote source is comprised of a pair of functions. One function retrieves a collection of data elements (with identifiers) from the remote store, allowing the data to be embedded and identified. The other function retrieves a subset of the collection retrieved by the first function that corresponds to a given set of identifiers.

Real-World Context: Real-world context refers to the real-world data (defined in Section 2.1) for which we want to gather insights using a Quill app. Real-world context could be an image of a receipt, a transcript, or a restaurant menu. The modality (e.g., image, text, speech) of the real-world context of an ARK-Join operation is specified in the QWL script. Each definition receives the incoming data elements and triggers corresponding data processing operations to execute the underlying ARK-Join operation as specified in the QWL script.

Embedding Store: Data elements from both real-world and remote sources are transformed into fixed-length vectors using a transformation function, called embedding functions (defined below) for an ARK-Join operation. Embedded data elements from a remote source are persisted in a data store, called the *embedding store*. A Quill embedding store comprises a remote data store that contains embedded data elements as fixed-length vectors and an index over those fixed-length vectors. We construct an index over data elements from remote sources for efficient implementation of K-nearest-neighbor-search for executing an ARK-Join operation on large-scale remote sources. We provide a detailed description of the ARK-Join operator in the following section. To construct an embedding store, a developer specifies the embedding function to be used and the indexing scheme as parameters in the QWL script. A Quill embedding store can be located in the local cache, a large-scale remote data store, or any suitable store reachable from the Quill app.

3.2 Transformation Functions

Quill supports two major transformation functions: *embedding functions* and *inline transforms*. Data elements from both *real-world context* and *remote sources* are transformed to a shared vector-space to execute the underlying ARK-Join operation. Quill utilizes an *embedding function* for this purpose. Data-elements r and s are joined together if $f_{(r)}$ and $f_{(s)}$ are semantically similar, where f denotes the *embedding function* that encodes data-elements into a fixed-length vector. Generalized *inline transforms*, on the other hand, are used to format the results returned by an ARK-Join operation to be presented back in the live camera view of an AR-enabled interface that the real-world data came from. We describe these transformation functions next.

Embedding Functions: Quill utilizes embedding functions to transform data elements from the *real-world context* and *remote source* to a fixed-length vector in a shared vector space for the purpose of identifying if a real-world data element is semantically similar to a data element from a remote source. If they are similar, a join operation is executed between the two data elements, and the results are returned back. Quill takes embedding functions for each modality of real-world context and remote source used in an app.

Inline Transforms: Inline transform functions are responsible for taking the intermediate results returned by an ARK-Join operator and transforming them into a representation that is surfaced back to the client interface. For example, a built-in inline transform function can take the results of an ARK-Join operation as a dataframe and convert it into a JSON format that is more suitable for representing the results as a table in the live camera-view of an AR-enabled client interface.

3.3 The ARK-Join Operator

The ARK-Join operator performs a left-outer join between data elements from a *real-world context* and a *remote source*. Data elements from the remote source have already been transformed and stored as fixed-length vectors in an *embedding store*. As a result of this operation, data elements from the *real-world context* (i.e., the left operand) are joined with semantically similar data elements (i.e., the right operand) in the embedding store. Formally, we define the join predicates of an ARK-Join operation as the $K - n$ *approximate nearest neighbors* of the left operand among the set of right operands, where $0 \geq n \geq K$ is the number of approximate nearest neighbors that are not within similarity threshold θ. The result-set of this join operation is as follows.

$$\{K\text{-}ANN_{\theta}(x, embed_r(R)) \ : x \in embed_l(L)\} \tag{1}$$

Where L and R are the left and right operands, the embedding functions $embed_l$, $embed_r$, and θ and K are specified in a QWL script by the Quill app developer. This join operation retrieves $K - n$ data elements from the remote source R that are most similar to the real-world data element l. The use of approximate nearest neighbors in conjunction with the similarity threshold θ ensures that each left operand is matched with similar tuples, or the null set when no similar tuples (according to θ) are found. Therefore, it can be thought of as a left outer join on real-world data. Results of this join operation are formatted using *inline transform* functions before being returned back to the user. Contemporary researchers have shown that users have trouble tracking more than four or five objects at a time [17] in a camera-enabled interface. As we expect the number of incoming requests in the form of data elements from a *real-world context* to be low during each session, it is computationally reasonable to embed real-world data elements on the fly prior to the execution of the join operation. The number of data elements from *remote sources*, on the other hand, can be huge, which motivates the need of constructing an index over the precomputed embedding vectors for each data element in the *embedding store*.

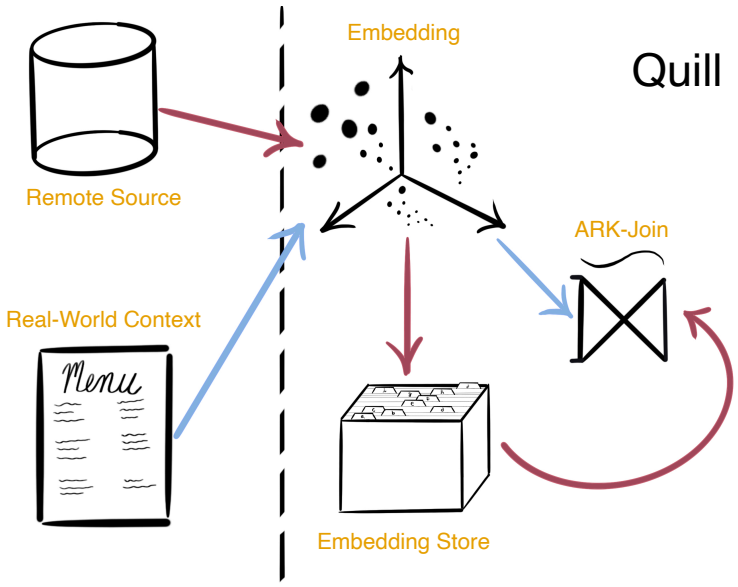


Figure 3: A high-level view of the grammar used by Quill's development framework. The red color arrows indicate the path data takes from *remote sources* to an *ARK-Join*, while the blue color indicates the path for data from *real-world context*.

3.4 QWL: Specifying an ARK-Join using a DSL

Internally, scripts written in QWL define a directed acyclic graph (DAG) between nodes that represent various event-driven elements of the grammar described above. The edges represent dependency relations amongst the nodes. Each node in the graph defines its own execution plan which either: (a) produces data if it is a *data source*-type element, or (b) transforms data if it is a *transformation function*-type element. The execution plan for a node is evaluated when either input from a client triggers the evaluation, or a node on which its query plan depends on has been evaluated. We describe how a QWL script defines relations between different grammar elements next.

QWL relates different grammar elements by using a function application construct, called *waited application*. A waited application lazily evaluates a node's execution plan based on when the nodes it depends on are available (i.e., not waiting) and all given conditions are met. For example, the evaluation of a node representing a *real-world context* can start after an AR-enabled client interface has detected a gesture-based query that sends data that should be joined to the Quill app in a request. By default, nodes in the DAG that are of *real-world context*-type are waiting, and thus all their dependents are waiting as well. A waited application relates data sources and transformation functions by taking both as inputs and evaluating the results after applying the transformation functions to the data sources. In QWL we use the `given` function to achieve the functionality of a waited application. Fig. 4 shows an example QWL script. Fig. 5 shows it corresponding DAG representation. `menu_db`, `menu_embedding`, and `menu_in_the_wild` are all source nodes that have no dependencies. Upon starting up the Quill app, Quill will evaluate all the nodes that are not waiting for an outside trigger or depend on a waiting DAG node. In this example, `menu_db` and `menu_embedding`, of type *Remote Source* and *Embedding* respectively, are source nodes that do not wait on a trigger so they will be evaluated. Quill then will evaluate the dependents of these nodes if there exists no nodes they depend on that are waiting. In this case, `embedded_menus` depends on `menu_db` and `menu_embedding` (defined by a *Waited Application*) and no other nodes that are waiting, so `embedded_menus` is evaluated. Since `join_menu` (an *ARK-Join* operator node) depends on `embedded_menus`, Quill will try to evaluate `join_menu` next. However, as `join_menu` also depends on `menu_in_the_wild`, which

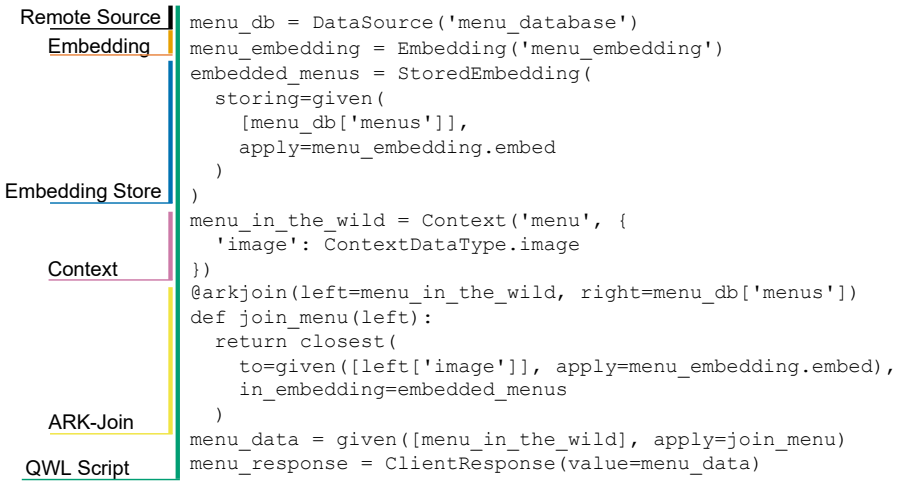


Figure 4: A snippet of the Python QWL interface for specifying Quill applications. This example uses an ARK-Join to retrieve data on a menu from a remote data source that corresponds to a real-world menu instance.

is a waiting node of type *real-world context*, `join_menu` will not be evaluated immediately. Once incoming data-elements are sent to the endpoint of the Quill app, represented by the `menu_in_the_wild` declaration ("/menu" in this case), the `menu_in_the_wild` node and its dependents will be evaluated. This includes the `join_menu` node that has no dependencies that are waiting. This leads to the `menu_response` node being evaluated which triggers a response to the client interface that contains the results of the ARK-Join operator `join_menu` defined in this QWL script. We discuss how an application is produced from a QWL script by describing how Quill's development framework is implemented in the following section.

4 System Design

4.1 From QWL to Quill App

A developer defines the specifications of the underlying data enrichment task using QWL. Quill's development framework takes a JSON representation of the QWL DAG of grammar elements described in Section 3.4 as input. This makes it easy to provide different dialects of QWL by developing an interface that produces this intermediate representation. Figure 4 shows an example Python QWL dialect. We represent the QWL DAG as a partially ordered collection of top-level declarations, called `Environment`. Internally, each declaration contains an identifier, an expression (corresponding to an element from the grammar in Section 3), the value of the expression's last evaluation (if it is not *waiting*), and the list of references to declarations that wait on it.

4.1.1 Execution and Runtime

Quill's execution engine begins by parsing the QWL script to construct an internal representation of the DAG described in Section 3.4. We refer to this internal DAG representation using the functional application construct `Environment`. For each declaration of a *real-world context* (defined in Section 3), Quill creates a server endpoint that the Quill app client can send requests (in the form of real-world data elements) to. Once an `Environment` is parsed, it is stored in the application server. We use a Redis store for persisting an `Environment`. The Quill app (defined in Section 2.1)), which is a server application, then listens for incoming requests in the form of real-world data elements from the client interface. A Quill app client acquires input data (`Context` in Figure 4)

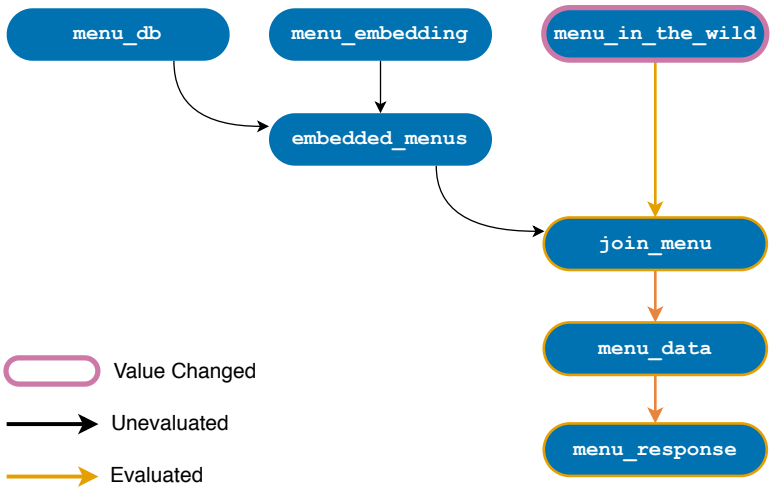


Figure 5: This figure presents the dependency sub-graph relevant to the QWL code presented in Figure 4. The graph shows the evaluation that occurs when data from the `Menu` is posted from the client. Quill evaluates declarations that depend on data from the `Menu` once the input data is posted.

and posts it to an endpoint on the server. Quill loads the `Environment` for the application and evaluates the declarations defined within it. Evaluation of the `Environment` is recursive, following the dependency structure specified in the DAG. In our implementation, we perform a topological sort of the expressions within the DAG to determine the order of evaluation. Once an ARK-Join operation is executed, the joined results are returned to the client. This is facilitated by the *waiting* application construct `ClientResponse`. A Quill app client receives a response when `ClientResponse` evaluates to a result set.

4.2 The Quill App Server

The primary function of a Quill app server is to manage the execution of each client session's `Environment`. Quill reevaluates each `Environment` when a data element from the real-world context is sent to an endpoint on the application server. Quill serializes and stores each `Environment` in external storage, indexed by a session ID. We use Redis to store `Environments` in our implementation. Data tied to the session state is stored in the state of the session's `Environment`. Due to the constraints posed by the AR-enabled client interface, we assume that the join results are of reasonable size, thus they are materialized and stored in the session state. If a developer wishes to store join results externally, Quill allows them to supply custom storage and retrieval functions as well. We describe our implementation of storing the fixed-length vectors representing each data element in a remote source for an ARK-Join operation next.

4.2.1 Implementing the Embedding Store

The ARK-Join operator takes transformed data elements from the real-world context as its left operand and transformed data elements from the remote source as its right operand. As mentioned in Section 3, a transformation function embed a data element as a fixed-length vector and stores it in the *embedding store*. The embedding store of a Quill app is shared across user sessions.

In order to efficiently perform an approximate nearest neighbor search during the execution of an ARK-Join operation, we adopt the inverted file system with asymmetric distance computation (IVFADC) [18] framework for indexing the embedded vectors of the remote source in an *Embedding Store*. This indexing method uses a coarse quantizer that assigns vectors to their corresponding elements in the inverted file system. Each element

in the inverted file system contains a set of similar vectors quantized using a fine quantizer. When querying, the query vector is assigned an element in the inverted file system, and then compared to the quantized vectors within the same inverted file system element. In our implementation, we use Redis as the external key-value store for the elements of the inverted file system. Quill makes three indexing options available to a developer out-of-the-box. They are as follows: (a) a locality-sensitive hashing (LSH) based method that uses binary sequences produced from locality-sensitive hashes as the coarse quantizer with no fine quantization, (b) IVF which is a k-means based method [18] as the coarse quantizer, with no fine quantization, and (c) IVFPQ which is a k-means based method [18] as the coarse quantizer, with product quantization as the fine quantizer. Quill also allows developers to define custom indexing methods within the IVFADC framework by defining custom coarse and fine quantizers.

5 Walk-through of an Example Use-Case

To demonstrate the generalizability of the Quill framework we undertake four separate case studies by re-implementing existing open-source applications using Quill. To investigate the usability of the QWL domain-specific language, we had three student developers each develop an application of their choice using Quill. We describe the applications in Table 5. In this section, we demonstrate the ease of development with Quill by providing a step-by-step walk-through of developing a Quill app for a real-world use-case scenario.

The Task: The main objective of the restaurant retrieval task is to enrich the live camera view of a dish in an AR-enabled interface with Yelp reviews for similar dishes from nearby restaurants. As described in Section 3, to develop this application, we need to define the following grammar elements: *real-world context*, *remote source*, and the *transformation functions* to be used by the underlying ARK-Join operation.

The Grammar Elements: The *real-world context* of this application is food images captured by the live camera view of the Quill app client. The *remote source* is Yelp entries of nearby restaurants. Lastly, the *transformation functions* are learned embedding methods [19] that align food images with text description in Yelp database.

QWL Specification: With the prerequisites in order, we can write our QWL specification. We start by specifying the function that is responsible for gathering a collection of data elements from the remote source. For this application, we can use Yelp's publicly available API for this purpose. Then, we specify the input transformation function to be used for embedding the data elements from the remote source. We can use a learned embedding function such as Carvalho et al. [19] that employs a cross-modal neural network to transform unstructured text into a shared vector space. Next, we specify the indexing method that we are going to use for the embedding store. Finally, we specify that the real-world data for this application is going to be an image. We can use the same transformation function for the real-world context. Once completed, the QWL script for this application should look similar to the script shown in Fig. 4.

From QWL to Quill App: With the QWL application configured and specified, the application server is ready to be started. We execute the command ``quill init'' from the Quill project directory, which transforms and stores data that is global to all sessions. Additionally, the base environment is stored for the given project. To start the application server, we execute the command ``quill run'' from the project directory, which produces a Quill app that accepts images of dishes from an AR-enabled camera-view and enhances it by surfacing recent customer reviews from nearby restaurants of similar dishes.

The Quill App: As we describe in Section 3.4, a Quill app is represented as a DAG of grammar elements. Here we describe how each grammar element in this example materializes in the actual app. The *real-world context* is represented as an HTTP POST endpoint that real-world data that should be joined is sent. This endpoint exists on

APPLICATION	REAL-WORLD CONTEXT	DATA SOURCES	EMBEDDINGS
Image Search	Image	Tiny Imagenet DB	VGG CNN
Facial Similarity	Face Image	Casia WebFace	Openface
Book Summarization	Book Title	Project Gutenberg	doc2vec
Movie Recommendation	User's ratings	MovieLens 100K	Movie Ratings
Restaurant Recommendation	Restaurant Name	Yelp Reviews	Transformer (BERT)
Coin Valuation	Coin Image	cointrackers.com	CNN/OCR output
Finding Bike Parts	Bike Image	Bike Database	Mutlimodal Triplet Network

Table 5: A table of applications implemented in Quill. The datasets pertaining to the Data Sources for Existing applications are Tiny Imagenet [20], CASIA WebFace [21], Project Gutenberg [22], and MovieLens 100K [23].

a server that the Quill app is hosted on. The *remote-source* is represented as an external data source and an index of the data, that can be stored external or internal to the server on which the Quill app is hosted on. All other *transformation functions* exist on the server the Quill app is hosted on.

6 Experiments

We evaluate Quill in terms of usability (is development with Quill easy?), performance (are applications developed with Quill performing acceptably?), and modularity/adaptability (can Quill adapt to advancements surrounding real-world data enrichment?). To evaluate performance, we compare the latency of the original implementations of existing applications described in Table 5 with the Quill implementations. We then discuss how to address retrieval accuracy while using Quill and the accuracy vs. latency trade-off. Our evaluation of usability considers how succinct and expressive the DSL is, as well as how usable it is as reported by the student developers who developed applications using it. The usability is also proven in the succinctness of scripts written in our DSL. We demonstrate the adaptability of Quill by utilizing embedding functions learned from Ember [8], a separate framework for keyless joins, and showing that Quill performs comparably to the keyless join from Ember, with added flexibility.

Applications: Quill's performance is evaluated against four existing applications, while the usability of Quill is evaluated using three novel applications developed by student developers. The applications are summarized in Table 5. Each application aims to complete a task that could be adapted to a multi-modal AR context. The image search application joins images from possibly two different modalities to return relevant images. The facilial similarity application is similar but restricted to facial images. The Book summarization application joins a book title (possibly obtained via the digitization of a book cover in the wild) and a topic with the full text of that book in order to provide of summary. The movie recommendation application joins a set of user ratings of movies with a dataset of movies and previous ratings in order to recommend movies to the user. The restaurant recommendation application joins a restaurant name (possibly digitized from the real world) with Yelp reviews in order to find similar restaurants in the area. The coin valuation application joins a coin image captured from the real world with a database of coin values to predict the value of the coin. The finding bike parts application joins a bike image with a bike database to let a user know where to find replacement parts.

Experimental Datasets: To compare the Quill implementations of existing projects to the original implementations, we use the data sets mentioned in the documentation of the projects obtained from GitHub when possible. The datasets used for measuring latency in each existing project are listed in the Data Sources column

of Table 5. To measure accuracy using the book summarization application, we use the WikiPassageQA [24] question answering dataset.

6.1 Performance Evaluation

Quill should produce apps that return data that enriches the real world promptly to enable the development of apps that can interactively enrich the real world. Consequently, Quill's performance evaluation is concerned with the latency of ARK-joins and how latency is affected at scale. Additionally, applications built with Quill should provide relevant results to the user. As a baseline for performance, we use the open-source projects described above. We describe the Quill implementations of these projects before presenting their performance evaluation results.

Implementations: The process described in Section 5 was used to implement each of the existing projects in Quill. For all the implementations, we use the IVF indexing method described in Section 4.2 to store embeddings. The number of centroids used in the indexing method is set to the square root of the number of stored data instances. The locality of the embedding stores relative to the machine running the Quill app was made to match the locality of the embeddings used in the original implementations.

Latency: For each application, we measure latency as a function of the size of the data sets we join against. Figure 6 shows results. Our goal is to demonstrate that applications built with Quill perform ARK-joins more efficiently and in a more scalable manner than those built without Quill. For each experiment, we select a random sample of the desired size from the corresponding data set for use by the baseline and the Quill implementation.

Figure 6 shows the latency of the original (or baseline) implementations of the applications compared to Quill implementations. For reference, the plot for the facial similarity application contains a line for the Quill implementation using a "full probe", meaning that a single centroid was used across the board, which equates to an exhaustive search on the embedded vectors. Quill beats all the baselines at scales greater than 10,000 data instances and grows at rates slower than those observed in the baseline implementations. The improvement in performance at scale comes from our use of indexing for ANN search, which processes only a portion of the data set instead of all data instances. However, we can see from the full probe results of the facial similarity example that even when a true nearest neighbor is retrieved instead of an approximate one, Quill outperforms the baseline. The baseline outperforms Quill on some very small data sizes, which is due to overhead surrounding embedding stores that would not be necessary for data sizes this small. Quill largely outperforms the book summarization baseline due to non-optimal programming practices leading to unnecessary computation such as training the embedding function every time a query is made. Using our DSL, developers can more reliably produce performant applications by avoiding anti-patterns such as retraining. Additionally, users benefit from the approximation modules built into Quill that can otherwise be difficult to utilize.

Accuracy: When developing an application that joins on the real world, it is paramount that relevant results are presented. Real-world decisions made with irrelevant information are fated to have negative or unexpected real-world consequences. The requirement for accuracy goes hand in hand with the requirement for latency. Joined results returned after the time they are needed are useless. Quill enables developers to tune the trade-off between latency and accuracy in their applications. Quill does this by exposing indexing parameters through the configuration file for the Quill application (project.yaml) or the embedding store expression of a QWL specification. For example, the IVF indexing method is parameterized by the number of centroids used for clustering. Along with the indexing configuration, the quality of retrieval depends on the embedding function and distance metric used. Quill enables developers to integrate customized versions or use off-the-shelf methods seamlessly in its framework.

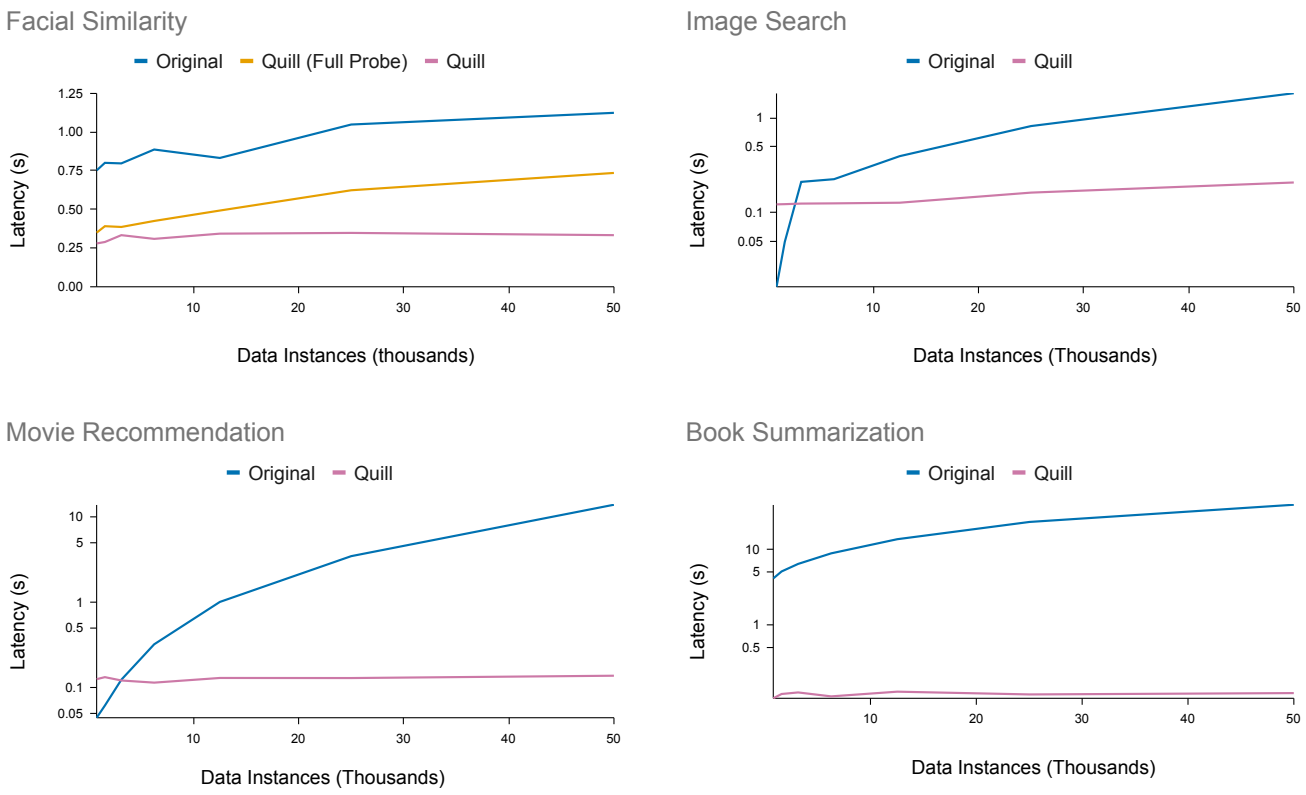


Figure 6: For each of the four existing projects we implement, we record the latency as a function of data size.

We demonstrate the change in accuracy with respect to the number of centroids for the IVF indexing method in Figure 7. We use mean average precision, mean reciprocal rank, and recall at 5, 10, and 20 when evaluating the accuracy. The retrieval quality metric that is important is application dependent, and these metrics cover a wide spectrum that allows us to demonstrate that ARK-Join accuracy is tunable within a Quill app. We measured the retrieval-based metrics by adding a `Context` expression to the book summarization application that takes in the number of desired centroids and utilizing that number in the embedding store expression to configure the number of centroids. As expected, as the number of centroids increases, the quality of retrieval goes down.

6.2 Modularity

The foundation of Quill is the ARK-join-centered grammar we defined in Section 3. This grammar should be modular enough to ensure that Quill can adapt to new technologies surrounding real-world joins, such as new embedding techniques. Thus to demonstrate the modularity of Quill, we evaluate the performance of a passage retrieval application developed in Quill that utilizes an embedding function from a state-of-the-art framework for keyless joins, Ember [8]. By showing that Quill can utilize this embedding method while outperforming a state-of-the-art keyless join, we establish that Quill is modular.

Using an open-source implementation of Ember, we compare the latency of the Ember keyless join with that of the ARK-joins from two Quill applications: one with a low latency cache and another without a cache. We measure latency on the MS MARCO passage retrieval dataset which contains a set of passages obtained from web pages pertaining to a sample of Bing queries [25]. The Ember keyless join and the Quill app without a cache both assume a 40 ms cloud-store communication latency. The caching Quill app simulates an application deployed in a local caching architecture similar to Figure 2, and doesn't incur the communication latency when the cache

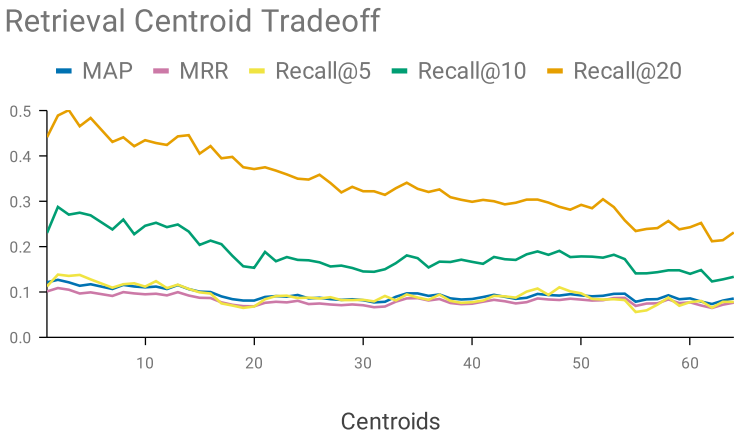


Figure 7: Using the book summarization application on the WikiPassageQA dataset we measure mean average precision (MAP), mean reciprocal rank (MRR), and recall at 5, 10, and 20 as a function of the number of centroids used in the IVF indexing of the embedded data.

is utilized. We configure the caching Quill app to use a similarity threshold that produces a cache hit ratio of 0.2. Because we are using identical embedding functions (obtained from Ember), we do not include the time to embed in our profiling of the join functions, which explains the difference in latency measurements observed in Section 6.1 and the measurements observed here. Figure 8 shows results.

Discussion: We note that Ember's keyless join uses an exhaustive maximum inner product search, while we use an approximate search. This results in the linear growth in latency for Ember that is not seen by Quill. The same accuracy and latency would be seen from Quill when the indexing of the embedding store is configured to be exhaustive. This emphasizes the flexibility of Quill to accommodate different accuracy and latency requirements. The low latency achieved by the caching Quill app emphasizes Quill's ability to adapt to different architectures for further performance improvements.

By abstracting the embedding function as a user-provided function, Quill can utilize future advances surrounding real-world data enrichment. This is demonstrated in our evaluation here which uses an embedding function learned from a state-of-the-art framework for keyless joins. Quill seamlessly utilizes this embedding function while providing flexibility with respect to performance and application architecture.

6.3 Usability

Quill provides a development framework that alleviates the effort currently required to produce an AR application that enriches real-world data. In order to measure the usability of Quill's DSL, we perform user studies by asking student developers to write an application of their choosing using a Pythonic dialect of QWL. We recruited three college students between the ages of 19 and 28, ranging in Python proficiency from little to significant experience. Two out of these three student developers had beginner-level experience with various concepts of applied machine learning. We measure the overall usability of Quill's DSL via results from the system usability scale (SUS) [26]. The SUS consists of 10 Likert scale questions and is an industry standard for measuring system usability. We also report findings on the succinctness of the Quill implementation of existing projects.

Test Setup: Each student was provided a one-on-one tutorial on QWL. The content presented to the students was identical, minus any questions asked. At the end of the tutorial, students were asked to describe their

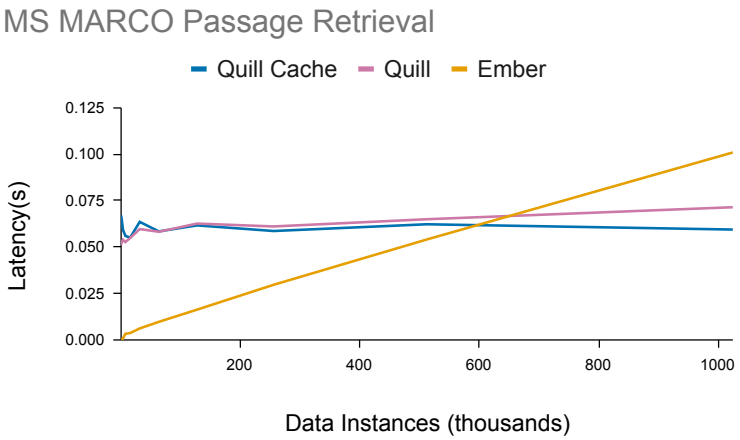


Figure 8: Using passages from the MS MARCO dataset [25], we compare the performance of the keyless join from Ember [8] with the ARK-join from two Quill applications: one with a low-latency cache, and one without a cache.

application using a DAG of elements from our grammar described in Section 3. The number of iterations required during this process is elaborated in the discussion. When a student was happy with their application description, and it was confirmed valid by the proctor, they moved on to writing the QWL code. We provided documentation and sample QWL scripts for reference. We asked students to bring a laptop with their text editor of choice to write the QWL code. The number of iterations required to produce a valid QWL script was recorded. Concluding the study, we asked students to complete the SUS survey.

User Study Results: Throughout our discussion of the user studies, we refer to the students as A, B, and C, who developed the coin valuation app, the restaurant recommendation app, and the bike parts app respectively. All of the students completed their valid DAG of grammar elements in a single iteration without help from the proctor. Like the DAGs, QWL scripts for all students' applications were completed in a single iteration without help. While questions were asked during the tutorials, there were no questions needed to create the DAGs or QWL scripts. Each student in this study took the system usability scale based on their experience with QWL. Results are shown in table 6. QWL scored an average of 68.33. Studies show that scores of 68 and above are considered good [26, 27], implying QWL scores well in terms of usability. In addition to the SUS results, we found that the Quill implementations of existing projects used in our performance evaluation were more succinct, using 30% less lines of code on average.

Maintaining Performance alongside Usability: The Quill framework must produce performant applications while remaining usable. In this section, we demonstrated that Quill is usable, and in Section 6.1, we showed that Quill produces performant applications. The balance between usability and performance is made possible by allowing the underlying framework to handle the complexities of applications that join real-world data. Quill prevents developers from falling into the traps of anti-patterns such as retraining; developers provide the embedding models, and Quill decides how to use them. Our implementation of the ARK-join handles many optimizations that lead to clumsy code when done by hand, such as indexing for approximate nearest neighbor operations. Moreover, Quill uses efficient approximation modules that can be difficult for the lay user to utilize.

QUESTION	AVERAGE
I would like to use this frequently	Agree
The system was easy to use	Neutral
The various functions in this system were well integrated	Agree
Most people would learn to use this system very quickly	Agree
I felt very confident using this system	Neutral
I found the system unnecessarily complex	Disagree
I would need the support of a technical person to use this	Disagree
There was too much inconsistency with this system	Disagree
The system is cumbersome to use	Disagree
I need to learn a lot of things before using this system	Disagree

Table 6: Results of a Systems Usability Survey on student developers. Results suggest that Quill scores well in terms of usability

6.4 Discussion

Our evaluation of Quill demonstrates its ability to simply define complex ARK-join functionality to produce robust, scalable applications. As demonstrated by the latency results of the image search application in Figure 6, it is possible that, for applications with small amounts of data, a non-Quill implementation can provide better results. Here it should be noted that Quill is more generalized than highly application-specific solutions that do not extend to other use cases. Optimizing to reduce latency for small data sizes is considered future work. After taking the SUS, we asked the developers to provide free form feedback about their experience using QWL. One developer expressed, "My only concern would be making the embedding fit the scope of the expected data so the ARK-Join can return proper results however that is not the goal of the project and relies on development from the user." This reiterates what we stated previously. Quill provides a flexible way to utilize off-the-shelf as well as custom embedding functions and indexing techniques in a single development framework, as the quality of retrieval is largely dependent on the embedding and indexing methods used by the developer. Another developer remarked, "I can not think of why I would use a non-closest neighbor join condition. But I can think of reasons to multiply the differences by a vector of attribute weights (i.e., non-euclidean distance)". This student refers to the way ARK-Joins are defined in QWL: the developer provides a predicate for the join, which is often a nearest neighbor-based predicate. While the student may be suggesting that we assume the condition will always be the nearest neighbor operation, we did not make this assumption to allow for classical join predicates.

7 Related Work

7.1 Interactive Data Systems for AR

The main idea of interactive data systems in AR revolves around bringing data to the real world in a way that allows users to enhance their view of reality. Earlier works in this domain, such as AR web-browsers, attempt to bring together data from multiple sources to create such an experience. Argon [28] is one such architecture that brings together multiple separately authored AR applications (built using their KHARMA [32] framework) into one browser view. While this platform does bring multiple data sources into the real world, it exclusively supports sensor and marker-based AR. Quill provides marker-less AR in which experiences are determined based on semantic relationships. The performance requirement of interactive data systems for AR poses a challenge to the development of such systems. Schneider et at. [29] utilize edge computing technologies to allow AR applications to offload some of the computationally heavy tasks associated with AR. Their work focuses on offloading tasks

common across AR applications, while Quill focuses on improving performance in the context of real-world data understanding and enrichment.

7.2 Keyless Joins

Recent works on keyless joins have proven them useful where similarity-based joins are needed and defining similarity functions explicitly is unreasonable. Ember [8] uses transformer-based machine learning techniques to learn similarity functions for joins used in context enrichment in machine learning pipelines. Termite [9] uses Siamese networks to learn similarity functions to aid in data integration tasks on heterogeneous text-based structured and unstructured data. These works learn keyless joins in frameworks that operate on modalities of data present in their respective tasks (ML context enrichment and text-based data integration). In contrast, Quill abstracts the similarity function of our keyless join (ARK-join) as a parameter to enable Quill apps that operate on varying modalities of data. Additionally, Quill's ARK-join can adapt to state-of-the-art learned similarity functions.

7.3 Semantic representation of relational data stores

The desire to link datasets based on *semantic* information has given rise to work that utilizes learned embeddings to represent structured data in a semantic vector space. The work titled Cognitive Databases [30] represents relational data as unstructured text in order to utilize word embeddings to represent the relational data as a vector. Similarly, Cappuzzo et al. [31] create embeddings of relational data for data integration by representing the data using a graph which allows them to use graph embedding techniques. We benefit from these works by utilizing methods they describe to embed relational data sets. Fernandez et al. [4] utilize word embeddings in order to find links within datasets semantically. In their recent work [9] called Termite, they used embedding techniques to perform data integration for various relational data stores. The additional constraint Quill imposes on its operation is the capability of handling multimodal real-world data as well as heterogeneous relational database schema within the interactive latency imposed by AR interfaces.

8 Conclusion and Future Work

Increasing advancement of technology in key related areas has made sure that widespread use of augmented reality (AR) is an inevitability rather than a distant possibility. While the necessary computing resources and technology needed to make immersive AR applications have become more readily available, developing a performant AR application that enriches real-world data with digital information is still a challenging task. Quill makes it easier to develop immersive AR applications that make it easier for the user to gather insights from real-world data by using an expressive DSL backed by an intuitive grammar. Quill serves as a natural extension of ARQuery [2] by enabling join queries specified using ARQuery to utilize remote data to enhance data in the real world. We demonstrate that Quill's development framework not only makes it easier to develop data-rich AR applications for developers with limited background knowledge in related technical areas, but it also produces more performant applications than other existing developer tools. Looking forward, we realize that although Quill accelerates the development workflow for AR applications, it only addresses a small portion of the problems surrounding the development of AR applications that enhance a user's view of real-world data. For example, sharing a user's enhanced view of real-world data within an interactive, AR-enabled interface to her friends on social media in a scalable way is an important piece of future work. To what extent an AR-enabled application should alter a user's view of the real world also requires a closer look at the underlying problem, as if done wrong it can have serious adverse consequences. Additionally, we hope to improve the performance of Quill apps by intelligently placing caching and indexing on the edge in a declarative manner similar to Shaowang et al. [33]. We

hope to address these problems in our future works as more advances are made in democratizing the development of AR applications empowered by a vibrant larger-than-ever developer community.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 1910356.

References

[1] Ronald Azuma, Yohan Baillot, Reinhold Behringer, Steven Feiner, Simon Julier, and Blair MacIntyre. 2001. Recent Advances in Augmented Reality. IEEE CG&A (2001).

[2] Codi J Burley and Arnab Nandi. 2019. ARQuery: Hallucinating Analytics over Real-World Data using Augmented Reality. In CIDR.

[3] Arnab Nandi, Lilong Jiang, and Michael Mandel. 2013. Gestural Query Specification. VLDB (2013).

[4] Raul Castro Fernandez, Essam Mansour, Abdulhakim A Qahtan, Ahmed Elmagarmid, Ihab Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2018. Seeping semantics: Linking datasets using word embeddings for data discovery. In 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 989–1000.

[5] Ritesh Sarkhel and Arnab Nandi. 2019. Deterministic Routing between Layout Abstractions for Multi-Scale Classification of Visually Rich Documents. In IJCAI. 3360–3366.

[6] Ritesh Sarkhel and Arnab Nandi. 2019. Visual Segmentation for Information Extraction from Heterogeneous Visually Rich Documents. In Proceedings of the 2019 International Conference on Management of Data. ACM, 247–262.

[7] Ritesh Sarkhel and Arnab Nandi. 2021. Improving information extraction from visually rich documents using visual span representations. Proceedings of the VLDB Endowment 14, 5 (2021), 822–834.

[8] Sahaana Suri, Ihab F Ilyas, Christopher Ré, and Theodoros Rekatsinas. 2021. Ember: No-Code Context Enrichment via Similarity-Based Keyless Joins. arXiv preprint arXiv:2106.01501 (2021).

[9] Raul Castro Fernandez and Samuel Madden. 2019. Termite: a system for tunneling through heterogeneous data. arXiv preprint arXiv:1903.05008 (2019).

[10] Peng Li, Xiang Cheng, Xu Chu, Yeye He, and Surajit Chaudhuri. 2021. Auto-FuzzyJoin: Auto-Program Fuzzy Similarity Joins Without Labeled Examples. In Proceedings of the 2021 International Conference on Management of Data. 1064– 1076.

[11] Foto N Afrati, Anish Das Sarma, David Menestrina, Aditya Parameswaran, and Jeffrey D Ullman. 2012. Fuzzy joins using mapreduce. In 2012 IEEE 28th International Conference on Data Engineering. IEEE, 498–509.

[12] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. 2011. Pass-join: A partition-based method for similarity joins. arXiv preprint arXiv:1111.7171 (2011).

[13] Ahmed Metwally and Christos Faloutsos. 2012. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. arXiv preprint arXiv:1204.6077 (2012).

[14] Zhimin Chen, Yue Wang, Vivek Narasayya, and Surajit Chaudhuri. 2019. Customizable and scalable fuzzy join for big data. Proceedings of the VLDB Endowment 12, 12 (2019), 2106–2117.

[15] Christopher D Manning and Prabhakar Raghavan. 2008. Introduction to Information Retrieval. Cambridge University Press.

[16] Jens Dittrich and Alekh Jindal. 2011. Towards a One Size Fits All Database Architecture. CIDR (2011).

[17] Patrick Cavanagh and George A Alvarez. 2005. Tracking multiple targets with multifocal attention. Trends in cognitive sciences 9, 7 (2005), 349–354.

[18] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. IEEE transactions on pattern analysis and machine intelligence 33, 1 (2010), 117–128.

[19] Micael Carvalho, Rémi Cadène, David Picard, Laure Soulier, Nicolas Thome, and Matthieu Cord. 2018. Cross-modal retrieval in the cooking context: Learning semantic text-image embeddings. In The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval. ACM, 35–44.

[20] Ya Le and Xuan Yang. 2015. Tiny imagenet visual recognition challenge. CS 231N (2015).

[21] Dong Yi, Zhen Lei, Shengcai Liao, and Stan Z Li. 2014. Learning face representation from scratch. arXiv preprint arXiv:1411.7923 (2014).

[22] Shibamouli Lahiri. 2014. Complexity of Word Collocation Networks: A Preliminary Structural Analysis. In Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics. Association for Computational Linguistics, Gothenburg, Sweden, 96–105. http://www.aclweb.org/anthology/E14- 3011

[23] F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. Acm transactions on interactive intelligent systems (tiis) 5, 4 (2015), 1–19.

[24] Daniel Cohen, Liu Yang, and W Bruce Croft. 2018. Wikipassageqa: A benchmark collection for research on non-factoid answer passage retrieval. In The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval. 1165–1168.

[25] 2021. MS MARCO. https://microsoft.github.io/msmarco/

[26] John Brooke. 1996. SUS: a "quick and dirty" usability. Usability evaluation in industry (1996), 189.

[27] Aaron Bangor, Philip T Kortum, and James T Miller. 2008. An empirical evaluation of the system usability scale. Intl. Journal of Human–Computer Interaction 24, 6 (2008), 574–594.

[28] Blair MacIntyre, Alex Hill, Hafez Rouzati, Maribeth Gandy, and Brian Davidson. 2011. The Argon AR Web Browser and standards-based AR application environment. In 2011 10th IEEE International Symposium on Mixed and Augmented Reality. IEEE, 65–74.

[29] Michael Schneider, Jason Rambach, and Didier Stricker. 2017. Augmented reality based on edge computing using the example of remote live support. In 2017 IEEE International Conference on Industrial Technology (ICIT). IEEE, 1277–1282.

[30] Rajesh Bordawekar, Bortik Bandyopadhyay, and Oded Shmueli. 2017. Cognitive database: A step towards endowing relational databases with artificial intelligence capabilities. arXiv preprint arXiv:1712.07199 (2017).

[31] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating embeddings of heterogeneous relational datasets for data integration tasks. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 1335–1349.

[32] Alex Hill, Blair MacIntyre, Maribeth Gandy, Brian Davidson, and Hafez Rouzati. 2010. Kharma: An open kml/html architecture for mobile augmented reality applications. In 2010 IEEE International Symposium on Mixed and Augmented Reality. IEEE, 233–234.

[33] Ted Shaowang, Nilesh Jain, Dennis D Matthews, and Sanjay Krishnan. 2021. Declarative data serving: the future of machine learning inference on the edge. Proceedings of the VLDB Endowment 14, 11 (2021), 2555–2562.

In-Database Decision Support: Opportunities and Challenges

Azza Abouzied,[⋆] Peter J. Haas,[◇] and Alexandra Meliou [◇]
[⋆] New York University (NYU) Abu Dhabi, United Arab Emirates
[◇] University of Massachusetts Amherst, USA

Abstract

Decision makers in a broad range of domains, such as finance, transportation, manufacturing, and healthcare, often need to derive optimal *decisions given a set of* constraints *and* objectives*. Traditional solutions to such constrained optimization problems are typically application-specific, complex, and do not generalize. Further, the usual workflow requires slow, cumbersome, and error-prone data movement between a database and predictive-modeling and optimization packages. All of these problems are exacerbated by the unprecedented size of modern data-intensive optimization problems. The emerging research area of in-database prescriptive analytics aims to provide seamless domain-independent, declarative, and scalable approaches powered by the system where the data typically resides: the database. Integrating optimization with database technology opens up prescriptive analytics to a much broader community, amplifying its benefits. In the context of our prior and ongoing work in this area, we discuss some strategies for addressing key challenges related to usability, scalability, data uncertainty, dynamic environments with changing data and models, and the need to support decision-making agents. We indicate how deep integration between the DBMS, predictive models, and optimization software creates opportunities for rich prescriptive-query functionality with good scalability and performance.*

1 Introduction

Prescriptive analytics [15, 19], and constrained optimization in particular, is central to decision making over a broad range of domains, including finance, transportation, manufacturing, and healthcare. In these settings, decision makers frequently face constrained optimization problems: they need to derive *optimal* decisions given a complex set of interacting *constraints* and *objectives*. Constraints arise from competition between activities for scarce resources such as time, budget, workers, trucks, tools, etc., and objective functions formalize organizational goals such as minimizing costs or delays, maximizing revenue, or minimizing disease mortality.

Optimization models rely on *predictive analytics*—using historical data to predict future trends as well as the future effects of current actions—in order to assess which actions will yield the best results. Predictive models can take the form of complex mechanistic simulation models that incorporate deep domain knowledge or data-driven models such as classical regression models or time series models or, more recently, machine learning models. Moreover, *descriptive analytics*—analyzing historical data to discover patterns and relationships—also plays a key role by informing the process of building optimization models so that they capture the most important relationships. Despite the fundamental interplay between descriptive, predictive, and prescriptive analytics, the latter has received much less attention from the database community.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Modeling and solving optimization problems has typically relied on application-specific solutions. Such solutions are often complex and do not generalize; a decision maker seeking to apply optimization techniques in a new application setting must either develop a new custom model from scratch, or must learn the intricacies of generic optimization software, which can be daunting for those with domain, but not optimization, expertise. Moreover, the usual workflow requires that data be extracted from a database and then reformatted and fed into a separate optimization package, after which the output must be reformatted and inserted back into the database; this process is slow, cumbersome, and error-prone. These challenges are further exacerbated by the unprecedented size of modern data-intensive optimization problems.

In-database prescriptive analytics is an emerging research area that aims to provide domain-independent, declarative, and scalable approaches, supported and powered by the system where the data relevant to these problems typically resides: the database. This makes modeling less ad-hoc, and the overall optimization process, from data preparation through solution and exploration of results, becomes much more efficient. Desirable data management functionality, such as efficient retrieval, consistency, persistence, fault tolerance, access control, and data-integration capability, become an integral part of the system "for free". Interest in native DB support for prescriptive analytics has therefore started to grow [13]. One line of research is exemplified by the SolveDB and SolveDB+ systems [24, 25]. These systems provide semi-declarative languages for specifying a broad range of optimization problems, allow easy sharing of optimization models across sub-problems of an overall predictive-analytics problem, and facilitate plugging in of various prediction models and optimization-problem solvers.

We have found, however, that existing solvers are often unable to deal gracefully, if at all, with very large amounts of data, with uncertainty in the data, with dynamic environments where the data or models are constantly changing, or with problems that involve finding optimal policies for automated decision-making agents. Thus, the naive use of black-box solvers is often not viable for modern large-scale optimization problems in complex environments. Our initial work aims to support the evaluation of an important class of constrained optimization problems—integer linear programs (ILP)—within a database [6, 7]. We have built a prototype system, PackageBuilder, to specify and evaluate ILPs as "package queries". As a simple example, each row of a database table might represent a food item, with attributes describing purchase price and nutritional content. A "package query" would return a "package" of rows (i.e., set of food items) having the minimum possible cost while satisfying constraints on minimum required nutritional content. More specifically, each food item i can appear x_i times in the package, where each "decision variable" x_i is a nonnegative integer, hence the "I" in ILP. Both the package cost, which is to be minimized, and the total amount of a nutrient such as vitamin D, which is constrained to lie above a threshold, are linear functions of the x_i decision variables, hence the "L" in ILP. Our emphasis has been on developing fully declarative SQL extensions to specify package queries over both deterministic and uncertain data, and to "open up" black box solvers in order to develop novel scalable approximate optimization algorithms for massive, possibly uncertain data in dynamic environments. Our work is complementary to that in [24, 25], in that our techniques can potentially be incorporated into a system such as SolveDB+.

ID	stock	type	price	···	gain
1	AAPL	Tech	150	···	?
2	MSFT	Tech	272	···	?
3	TSLA	Tech	758	···	?
4	AMZN	Tech	2400	···	?
5	INTC	Tech	42	···	?
6	GOOG	Tech	2280	···	?
7	ADBE	Tech	415	···	?
8	FB	Tech	194	···	?

Figure 1: Stock_Investments table.

The challenges that we consider are concretely illustrated by the following example.

Example 1 (Investment portfolio): A broker wants to construct an investment portfolio for one of her clients. The client has a budget of $50K and a planning horizon of six months. The available data comprises a table where each row corresponds to a stock. The attributes for a stock include its current purchase price per share, as well as other features such as industry sector, financial rating of the company, recent volatility, and so on; see Figure 1.

The "?" symbols indicate that the attribute gain, which represents the gain from selling a given stock six months from now, is uncertain. The broker has available a computer model that uses the stocks' characteristics, along

with other market data, to predict the joint probability distribution of stock prices six months from now. The broker wants to select a package of tuples, i.e., a portfolio of stocks, that will maximize the client's expected profit in six months, subject to the constraints that (1) the total purchase price of the portfolio does not exceed \$50K, and (2) the probability of losing \$1000 or more at the end of six months is at most 5%.

As stock prices change, new stocks are added, and prediction models get updated, the optimal portfolio package may also change. Further, as the broker discusses options with her client, they may wish to explore slight variations, e.g., *what if she slightly increases or decreases her budget? what if she increases or decreases the risk tolerance on her investment?* or, *what if she wishes to eliminate or include a specific set of stocks in her portfolio?*

This example highlights several challenges.

Query specification. In our example, while the current price of a stock may be known (deterministic), its future price is a random variable whose value can only be described via a probability distribution. The future value of a portfolio package is therefore also uncertain. Even in the deterministic case, standard SQL is incapable of expressing package constraints [6, 7]. In the stochastic setting, statistical concepts such as expected values, probabilistic constraints, and risk measures such as "conditional value at risk" (CVaR) compound the specification challenge. How does one model the uncertainty of stock prices and specify probabilistic constraints (such as bounding the risk of loss) or reason about stochastic objectives and variables? Especially challenging is the problem of specifying data uncertainty: probability distributions over uncertain data values take on many different forms, including discrete, continuous, and mixed distributions. Moreover, closed-form descriptions of probability distributions are not always available, e.g., for financial models of complex instruments such as "exotic" stock options.

Scalable processing. Because the number of decision variables in a package query equals the number of rows in a database table, which can run to the millions, the resulting ILP is often much too large for an off-the-shelf solver to handle, even in a deterministic setting. With uncertain data, Monte Carlo methods must often be used to create an approximating ILP whose size is larger than the ILP for the deterministic case by orders of magnitude, exacerbating the problem. Indeed, a variant of the above example allows stocks to be held for varying amounts of time before sale, so that each distinct stock may be represented by multiple rows in the table corresponding to different potential sell dates; this can inflate the size of the optimization problem by more orders of magnitude.

Dynamic environments. Changes to the underlying data may be frequent, incurring expensive recomputations of the query result to keep it up-to-date. Similarly, reevaluating the packages from scratch, even for small variations of the parameters, to explore different scenarios and options, can make a what-if analysis prohibitively time-consuming.

Policy-making As we move from one-off decisions to automated decision making by agents in time-changing and dynamic environments, the goal of prescriptive analytics shifts to construction of robust *policies* that inform decisions based on (partial) observations of the current world and predictions of the future world. Consider an auto-trader that makes daily decisions about stocks to buy or sell in order to optimize for long-term gain. The auto-trader employs an *optimal* policy *learned* from stock-price time-series predictions. As decision-making becomes increasingly autonomous, with sequential decisions in dynamic and uncertain environments, predictive and prescriptive analytics become tightly enmeshed, with *reinforcement learning* methods coming into play and requiring better data management support for the massive simulated or real data sets used to constantly train predictive models, learn optimal policies and model possible outcomes.

In the following sections, we consider the various challenges mentioned above for in-database specification and solution of package queries. The discussion focuses on our prior and ongoing results as well as indicating directions for future work.

2 Query Specification

To allow declarative specification of package queries in a DBMS-friendly manner, a natural approach is to extend the SQL language to support such queries. We first introduce PAQL, our SQL extension for declarative specification of deterministic package queries, and then discuss further extensions to handle data uncertainty, leading to the SPAQL language extension.

2.1 Package Query Language (PaQL)

The PAQL language extension allows users to declaratively express combinatorial optimization problems with linear objectives and constraints natively within a relational database.

Example 2 (Meal Plan): A dietitian needs to design a meal plan for a patient. She wants three distinct gluten-free meals, between 2K and 2.5K calories in total, and with a low total intake of saturated fats. Each row in the relational table `Recipes` describes a distinct meal, giving the total number of calories and total amount of saturated fats and gluten, along with other nutritional information.

Package queries extend traditional database queries by allowing users to not only express standard selection constraints (e.g., each meal must be gluten-free), but also higher-order *package-level* constraints (e.g., all meal plans must have between 2,000 and 2,500 calories in total) and objectives (e.g., minimize the meal plan's total saturated fat). The Package Query Language (PAQL) extends SQL to allow for their declarative expression. The Meal Plan query in Example 2 can be expressed in PAQL as follows:

```
SELECT     PACKAGE(*)
FROM       Recipes REPEAT 0    /* each meal can appear at most once in a package */
WHERE      gluten = 0
SUCH THAT
           COUNT(*) = 3 AND
           SUM(kcal) BETWEEN 2.0 AND 2.5
MINIMIZE   SUM(sat_fat)
```

The PackageBuilder system transforms a PAQL specification into an ILP and uses an off-the-shelf ILP solver to compute the desired package. When, as is typical, the number of database rows is large, direct solution by the solver is infeasible because of the large size of the ILP. We have developed an iterative approximate solution algorithm called SKETCHREFINE [7] (discussed in Section 3) to handle large numbers of rows while providing approximation guarantees.

2.2 sPaQL

In preliminary work [8, 9], we have started to extend PackageBuilder to handle uncertain data. Dealing with uncertainty is crucial, because uncertain data is pervasive. For example, data values might represent outputs of predictive stochastic models as in the stock examples, or might derive from noisy processes such as sensor readings, privacy shielding, data integration, or extraction of structured data from text, images, or video [28]. Thus, decisions must often be made in the face of uncertainty, i.e., the decision maker must solve a *stochastic optimization problem*. As a first step, we have extended PAQL to allow specification of *stochastic package queries (SPQs)*; the extended language is called SPAQL. For example, the portfolio-selection problem of Example 1 can be expressed in SPAQL as follows:

```
SELECT     PACKAGE(*) AS Portfolio
FROM       Stock_Investments
WHERE      type = 'Tech'
```

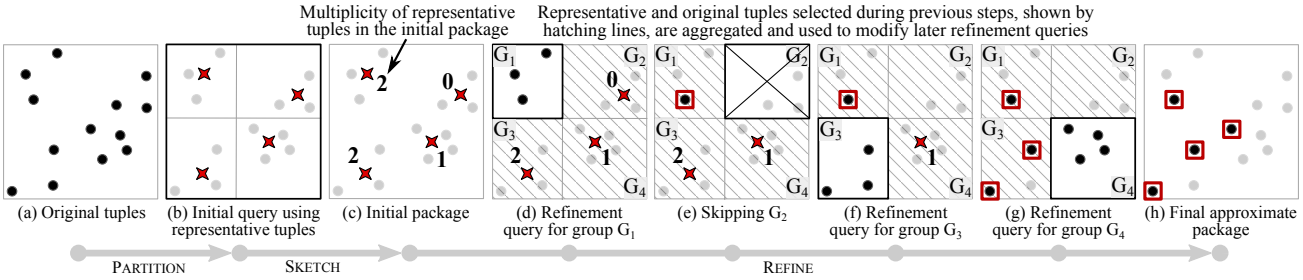


Figure 2: An illustration of the SKETCHREFINE algorithm. The original tuples (a) are partitioned into four groups and a representative is constructed for each group (b). The initial sketch package (c) contains only representative tuples, with possible repetitions up the size of each group. The refine query for group G_1 (d) involves the original tuples from G_1 and the aggregated solutions to all other groups (G_2, G_3, and G_4). Group G_2 can be skipped (e) because no representatives could be picked from it. Any solution to previously refined groups are used while refining the solution for the remaining groups (f and g). The final approximate package (h) contains only original tuples. (Figure taken from [7].)

SUCH THAT
SUM(price) $\leq 50{,}000$ AND
SUM(gain) ≥ -1000 **WITH PROBABILITY** ≥ 0.95
MAXIMIZE **EXPECTED** SUM(gain)

A SPAQL query such as the one above specifies a *stochastic* ILP (SILP) whose solution is the desired package. Expected-value constraints such as EXPECTED SUM(attr) $\geq x$ are also allowed.

3 Scalable Query Evaluation

In this section, we provide an overview of scalable evaluation methods for package queries (PQs). Scalability challenges arise in the presence of large datasets and are further exacerbated by the use of Monte Carlo methods for solving stochastic package queries (SPQs); the latter methods require generation of multiple "scenarios". We discuss the SKETCHREFINE algorithm for addressing the first challenge in the context of deterministic PQs and then discuss an algorithm called SUMMARYSEARCH for scalability with respect to scenarios. We then outline the challenges of bringing these two research threads together to evaluate SPQs over large sets of uncertain data.

3.1 Scaling Deterministic Package Queries with SketchRefine

The PackageBuilder system transforms a PAQL specification into an ILP and uses an off-the-shelf ILP solver to compute the desired package. When, as is typical, the number of database rows is large, direct solution by the solver is infeasible because of the large size of the ILP. In prior work, we developed an iterative approximate solution algorithm called SKETCHREFINE [7] to handle large numbers of rows while providing approximation guarantees. Briefly, the algorithm first partitions the rows into groups, where the rows in each group have similar attribute values, and then computes a representative for each group. A small ILP using only the representatives can be then easily solved—the "sketch". The sketch is then iteratively "refined" by carefully replacing each representative using the rows that it represents. The process maintains feasibility of the current solution until the final package is obtained. Limiting the size of each group to be a small number of τ tuples ensures that each refine phase can be executed efficiently and allows for approximation guarantees. Figure 2 illustrates the three key operations of SKETCHREFINE: partition, sketch and refine.

Scenario 1

ID	···	gain
1	···	20
2	···	3
3	···	-30
4	···	10
5	···	200
6	···	-10
7	···	30
8	···	20

Scenario 2

ID	···	gain
1	···	10
2	···	6
3	···	-5
4	···	4
5	···	120
6	···	-20
7	···	15
8	···	16

Scenario 3

ID	···	gain
1	···	-2
2	···	8
3	···	-25
4	···	36
5	···	70
6	···	15
7	···	20
8	···	7

Summary

ID	···	gain
1	···	-2
2	···	3
3	···	-30
4	···	4
5	···	70
6	···	-20
7	···	15
8	···	7

Figure 3: Three example scenarios and a summary for the Stock_Investments table.

3.2 Scaling Stochastic Package Queries with Respect to Scenarios

Evaluating SPQs is very challenging, due to the sheer size of the optimization problems: First, as with deterministic PQs, there is a decision variable for every row of a (potentially very large) table. Second, the presence of uncertainty typically requires multiple versions of the table, called scenarios, that represent Monte Carlo realizations of the uncertain data values. Often, a large number of scenarios is required for accuracy, inflating the problem size by orders of magnitude.

Exact evaluation: One approach tackles the scalability challenge of stochastic problems by eliminating uncertainty. E.g., it is indeed possible to translate a stochastic package query into an integer program when (i) the expected value of each attribute in an expectation objective is known and (ii) each attribute in a probabilistic constraint has a Gaussian distribution with known mean and variance. It is possible to derive precise rules that enable this translation. Consider, for example, an expectation objective on attribute A_j, i.e., the goal is to minimize the expected value of the sum of A_j over all tuples in the package. The objective function to be minimized takes the form $\mathbb{E}\left(\sum_i A_{ij} \cdot x_i\right)$. Because of the linearity of expectation, $\mathbb{E}\left(\sum_i A_{ij} \cdot x_i\right) = \sum_i \mathbb{E}\left(A_{ij}\right) \cdot x_i = \sum_i t_i.\mu\mathsf{A}_j \cdot x_i$, where $\mathsf{t_i}.\mu\mathsf{A_j}$ is the expected value of A_j for tuple t_i. Therefore, we can simply replace the expectation objective with a linear deterministic objective. Similarly, linear Gaussian probabilistic constraints can be transformed into deterministic quadratic constraints; we require, however, that the resulting constraints be convex.

Monte Carlo evaluation: The foregoing method has the great benefit of being exact, but has limited applicability and may fail even on small problems (few rows) because of the complexity of the quadratic constraints. In preliminary work, we have developed a more general strategy based on Monte Carlo (MC) sampling. The idea is to approximate the SILP with a deterministic ILP. Specifically, we replace the expectation objective with an empirical average over a set of randomly generated *scenarios*, and similarly replace a probabilistic constraint—e.g., that an inequality be satisfied with 90% probability—with a requirement that, e.g., the inequality hold for 90% of the scenarios. This approach forgoes exact optimality in order to (1) allow arbitrary distributions for random variables appearing in probabilistic constraints, (2) avoid the convexity requirements on probabilistic constraints mentioned previously, (3) allow correlation between tuples, and (4) obtain less complex (i.e., not quadratic) constraints. The only requirement is the ability to sample values from each random variable. To this end, we can store uncertain data in a probabilistic database [28], specifically a "Monte Carlo database system" such as MCDB or SimSQL [4, 10, 16, 17, 23], which offer support for arbitrarily complex random distributions of the kind needed to support optimization problems such as portfolio selection. Roughly speaking, the uncertainty of a given data item is specified via a user-defined *value generation (VG) function* which, when invoked, generates a sample realization of the data-item value from its underlying probability distribution. VG functions can generate values for multiple data items simultaneously, thereby allowing complex statistical correlations between data items. By evaluating every VG function in a table, we get a sample realization of the table, which we call a *scenario*. (Scenarios are also called "possible worlds" in the literature on probabilistic databases.) Figure 3 shows

three possible scenarios (the leftmost three tables) for the table in Figure 1. After computing an optimal package, its true feasibility and objective value for the original SILP can be determined to high accuracy using a very large number of "validation" scenarios; this calculation is much faster than solving the approximate ILP optimization problem.

The deterministic ILP obtained as described above is called a *stochastic average approximation (SAA)* and approximates the SILP. Often, many scenarios are required to obtain a sufficiently accurate approximation. Indeed, if too few scenarios are used, then the solution to the SAA will be infeasible for the true SILP, but will have an objective value that is better than the true one, so that we think that we are doing better than we actually are; this phenomenon is known as the "optimizer's curse" [27]. The size of the SAA is proportional to the number of database rows times the number of scenarios, which makes direct use of an off-the-shelf solver infeasible.

Scaling the number of scenarios: To address this scalability problem, we developed an algorithm called SUMMARYSEARCH [9] that can drastically reduce the number of scenarios required. SUMMARYSEARCH replaces the large set of scenarios used to form the SAA by a very small synopsis of the scenario set, called a *summary*, which results in a reduced ILP, called a *conservative summary approximation (CSA)*, that is much smaller than the SAA. A summary is carefully crafted to be "conservative" in that the constraints in the CSA are harder to satisfy than the constraints in the SAA, thereby pushing the solver to find truly feasible solutions. In our portfolio example, a summary of a set of three scenarios can be obtained by taking the row-wise minimum of the gains as shown in Figure 3; if the gain of a package with respect to the summary exceeds $-\$1000$, then clearly the gain of 100% of the original three scenarios exceeds $-\$1000$. By taking the row-wise minimum over more or fewer random scenarios, the summary can be made more or less conservative.

Because the ILP for the CSA is much smaller than that for the SAA, it can be solved much faster. Moreover, the resulting solution is much more likely to be truly feasible (as verified using a large set of validation scenarios), so that the required number of optimization/validation iterations is typically reduced. Of course, if a summary is overly conservative, the resulting solution will be feasible, but highly suboptimal. Therefore, during each optimization phase, SUMMARYSEARCH implements a sophisticated search procedure aimed at finding a "minimally" conservative summary; this search requires solution of a sequence of reduced ILPs, but each can be solved quickly. In experiments reported in [9], SUMMARYSEARCH was able to answer SPQs faster by orders of magnitude than the prior approach of sequentially adding more and more scenarios to an SAA until either the solution package is truly feasible (as measured by the validation set) or the solver chokes; indeed, SUMMARYSEARCH was able to compute good, truly feasible solutions in many cases where the prior approach would fail.

3.3 Fully Scaling Stochastic Package Queries

The SUMMARYSEARCH algorithm addresses the scalability issue with respect to the number of scenarios, but does not address scalability with respect to the number of database rows. For interactive decision making in dynamic, uncertain environments involving large amounts of data, both issues must be addressed simultaneously. In current work, we are extending our SPQ evaluation algorithms to handle large numbers of database rows. A promising direction is to use a sketch-and-refine approach as an "outer loop" to generate a sequence of SILP problems with a small number of rows which are solved using techniques that are scalable in the number of scenarios. To further improve speed and efficiency, a potentially powerful approach opens the lid on the ILP solver to try and exploit the characteristics specific to ILPs in the package-query setting.

Adapting and improving SketchRefine: As with SKETCHREFINE for deterministic data, we create the sketch by replacing the rows with a small set of representatives and then subsequently refine it. A key question is how to define the partitions and compute the representatives. In the deterministic setting, the distance between rows, which can be viewed as points in a multidimensional space S of attribute values, is relatively straightforward to define. In the stochastic setting, each row represents a multidimensional *probability distribution* over S. This raises the question of how to appropriately define distances between the stochastic rows. There are many distance metrics for probability distributions, and for any metric there is an trade-off between space requirements,

accuracy, and computational cost when estimating the metric from scenarios. Possible approaches include estimating a metric using raw samples, histograms, quantiles, or kernel-density methods [26]. It is also desirable to parallelize the partitioning operation, which cannot be easily done in the prior clustering-based approach used in SKETCHREFINE; we are therefore currently investigating alternative divide-and-conquer approaches. In addition, it is often the case that multiple data items use the same choice of VG function, perhaps with slightly varying parameterizations, to generate samples when creating a scenario; this information can potentially be exploited to develop effective partitioning schemes. Once a partition is determined, a (stochastic) representative can potentially be computed in a number of different ways, varying in accuracy and computational effort.

For the inner loop, we are investigating improvements to SUMMARYSEARCH by exploiting the tractability of *CVaR constraints*. The probabilistic constraint on the gain in the SPAQL query in Section 2.2 is called a *chance constraint* in the stochastic-programming literature. In risk-management terminology [20], the $\alpha = 0.95$ *Value at Risk* (VaR) is \$1000 in the worst case; that is, the probability of losing \$1000 or more is at most $1 - \alpha = 5\%$. Although VaR is a widely used risk measure, it has a number of deficiencies. Intuitively, knowing that there is at most a 5% chance of losing \$1000 *or more* is not totally reassuring, since the actual loss in the bad 5%-probability scenario—i.e., the "worst 5% of cases"—is not controlled at all. Risk analysts are increasingly preferring to supplement or replace the VaR measure of risk with an alternative measure, called the *Conditional Value-at-Risk* (CVaR)—also called *Expected Shortfall*—with confidence α, defined as the expected loss given that the loss exceeds the α-VaR. Unlike VaR, the CVaR measure has the desirable "subadditivity" property that bounds the total risk of a set of gambles by the sum of the individual risks, thereby encouraging risk reduction through diversification, unlike VaR.

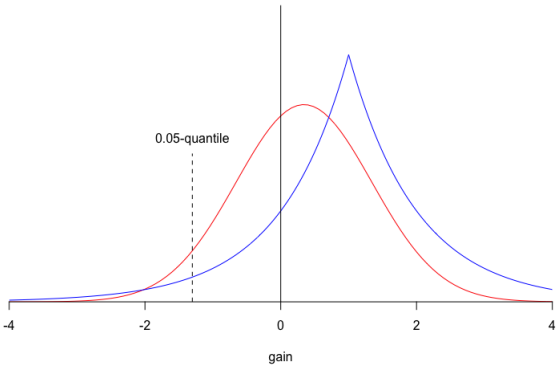


Figure 4: Two plots visualizing the probability distributions of the total gains of two different packages. The blue curve denotes a package with a higher expected sum, but in its lower 5% of cases, the average loss is almost three times higher than that of the red curve. VaR constraints alone do not differentiate between the two plots, since the right boundary of the lower 5% tail is the same in both distributions. CVaR constraints allow users to limit the expected loss in the tail of the distribution, and avoid high-risk packages such as one shown in blue.

As indicated in our discussion of exact evaluation methods in Section 3.2, a constraint defined in terms of an expected value can be converted to a simple linear constraint that is relatively easy to handle. In a similar manner, a CVaR constraint, since it comprises an expectation, can also be handled much more easily than a VaR constraint. Interestingly, it appears as if a problem with a VaR constraint can be solved by rapidly solving a sequence of problems involving only CVaR constraints, avoiding the need to search for appropriately conservative summaries.

Re-engineering the ILP solver: The ILP solver plays a key role in solving both deterministic and stochastic package queries, so speeding up the solver module can significantly improve the performance of all methods discussed so far. Moreover, in the setting of SKETCHREFINE, a solver that can scale to large numbers of rows reduces the number of refinement steps required and improves accuracy. An important characteristic of the ILP problems that arise in our setting is that the number of rows n is much larger than the number of constraints m, since constraints are typically manually defined by the user whereas the number of rows equals the table cardinality. This creates opportunities to replace a general-purpose ILP solver by a specially designed paralellizable approximate ILP solver (A-ILP), which can speed up processing not only of SPQs, but also of deterministic package queries and more general ILPs having a high n-to-m ratio. In preliminary work, we have started to develop such an improved solver. The rough idea is to first relax the ILP to a linear program (LP) by

dropping the integrality constraints. The LP is then solved using a novel variant of the well known dual-simplex algorithm [29] that exploits the n-to-m ratio in multiple ways for speed and efficiency. An $O(m + \log n)$ "guided lattice walk" is then used to obtain a reduced ILP with fewer variables that approximates the original ILP. The reduced problem is then solved exactly by a standard ILP solver. Initial experiments on an 8-core machine show that the A-ILP solver can have an accuracy comparable to the commercial Gurobi ILP solver while being twice as fast; the relative speed-up is expected to become even more pronounced as the number of cores increases due to the parallelizability of the novel dual-simplex algorithm. We also found that A-ILP scales well, approximately solving an ILP with tens of millions of variables in 5–8 seconds. We intend to theoretically analyze the properties of A-ILP and leverage it to enhance interactivity in PACKAGEBUILDER.

4 Dynamic Data and Models

Another key challenge for in-database optimization via package queries is that, even for small delta changes in the underlying data or in the query parameters, the computationally intensive package query needs to be re-executed from scratch. Yet, in most real-world applications, the overall form of the constrained optimization problem remains roughly the same, while the data or the query parameters incrementally change often within short time spans. The requirement to re-execute such queries from scratch makes maintaining results up-to-date computationally tedious and exploratory analysis impractical. In this section, we sketch some ideas for performing *incremental maintenance* of package results when the underlying data or query parameters change slightly in order to support interactive exploration and analysis. We discuss deterministic data first, and then extend our discussion to stochastic data.

4.1 Incremental PQ maintenance under data perturbations

Incremental package maintenance under data changes can be handled heuristically: Let P be the set of tuples appearing in a package result, R the set of tuples that were deleted from the original database D, and A the set of tuples that were added to D. A heuristic solution to the standing package query can be obtained by running it on the dataset $(P \setminus R) \cup A$. Assuming that the data change is small, this should be a small set of tuples, over which one can solve the constrained optimization problem directly. This method can serve as a heuristic baseline; a more principled approach rests on a modification of the SKETCHREFINE algorithm.

The SKETCHREFINE algorithm provides package solutions that are a $(1 \pm \epsilon)$-factor close to the optimal solution. It achieves this tight approximation bound by ensuring that each partition or group has a maximum diameter ω such that ω depends on ϵ and all tuples in the group are within a radius $\omega/2$ of the group's centroid or its representative tuple. Given this theoretical guarantee of SKETCHREFINE's behavior, we identify the *irrelevant tuples* of an incremental update as inserted tuples (ΔR) that can be placed into an existing group without violating its diameter ω or size τ limits, or deleted tuples (∇R) that do not occur in the solution package.

A scalable incremental SKETCHREFINE variant can potentially employ an incremental tree index to maintain tuple-group mappings and can easily split groups that exceed the diameter or size limits. In this way, recomputations are only triggered when an incremental update causes a tree restructuring. Moreover, since partitioning and representative tuple construction is incrementally maintained, only parts of the SKETCHREFINE algorithm need to be re-executed and not the entire algorithm for certain updates. For example, when a tuple appearing in the solution package is deleted from the base relation, we only need to re-refine the group to which the tuple previously belonged.

Additionally, one can potentially exploit the fact that the A-ILP solver described in Section 3.3 hinges on the dual-simplex algorithm. This algorithm is known to be amenable to incremental processing, and could potentially be leveraged for incremental package updating.

4.2 Incremental PQ maintenance under query perturbations

A decision-maker may wish to quickly determine the effect of changing some query parameters on the overall objective value. For example in the meal planning query (Example 2), *'does relaxing the constraint on calories lead to a lower-fat plan and to what degree*'? Often, a rapid, initial estimate of the objective value and an approximate, feasible, package solution is sufficient to help the decision-maker decide on whether they wish to settle on the new query parameters, in which case, the new query can be fully and accurately evaluated.

We identify five practical mechanisms for interactive query refinements: (i) changing the threshold parameters of base constraints to tighten or relax a query, (ii) tuple exclusions or tuple inclusions, (iii) changing the threshold parameters of global constraints, (iv) converting base constraints to global ones and vice versa, and (v) in the case of stochastic package queries changing the chance or CVaR parameters associated with a constraint. An interesting research challenge is to develop one or more scalable evaluation techniques that provide an upper-bound (lower-bound) estimate of the objective value for minimization (maximization) queries and an initial feasible package interactively. Heuristic methods can avoid making expensive calls to the integrated ILP solver and rely instead on efficient database top-k querying techniques. This is necessary to ensure sub-second response times that are crucial for exploratory interactive query refinement such as when a user drags a slider to control the threshold values of selection constraints: as they drag the slider, they would like to immediately understand the impact of their refinement on the objective value and overall package solution.

We briefly describe a promising heuristic, TOPKSKETCH, that we hope to expand and analyze. TOPKSKETCH computes and maintains an ordered list of tuples in each SKETCHREFINE partition: the ordering is determined by both the objective function and the selection constraints of the package query. The ordered lists are similar to an onion index [12]. On query refinement, we select the top-k tuples from each ordered list that satisfy the refined selection constraints. Along with the original package, we pass these new tuples to the solver to recompute a solution. This approach avoids the repeated refine operations (as a solution is built on actual tuples and not representative tuples) and is equivalent in complexity to a single sketch operation. In preliminary experiments, we found this approach to return results interactively (< 20 ms) for the refinements discussed above with near-optimal objective values.

4.3 Incremental SPQ maintenance

Perturbations to data and queries occur in the stochastic setting as well. For example, in the portfolio problem, an investor might originally only have considered NYSE stocks but may now also want to consider Nasdaq stocks. Should she replace some NYSE stocks with Nasdaq stocks in her previously optimal portfolio? The degree of risk than she might tolerate may change over time as her financial position or other factors change. Perhaps she wants to understand the effect on the expected gain from her "optimal" portfolio if interests rates turned out to be somewhat higher than the stock-prediction model originally assumed; is a modified portfolio needed?

The techniques discussed in Section 4.2, such as TOPKSKETCH, focus on incremental maintenance for deterministic package queries under small perturbations such as tightening or loosening constraints and adding or removing rows. Since these techniques essentially focus on updating the solution to an ILP, they are also potentially applicable, perhaps with modifications, for updating the solution to the types of ILP used in the various methods for SPQ evaluation. However, there are aspects of incremental maintenance that are unique to SPQs. In particular, even if the query and the set of stochastic database rows (e.g., stocks) stays the same, a user might want to experiment with the underlying probability distribution of the uncertain data values. In particular, a predictive stochastic model used to generate scenarios often involves a vector θ of parameters that a user might want to vary. For example, a predictive stock-price model might assume certain values for interest and unemployment rates, and the user might want to see how the optimal portfolio changes when these parameters are varied. If the parameters are not perturbed too much, are there good ways of leveraging prior optimization results?

Quick updates via stochastic search: When the parameter changes are relatively small, it may be advantageous

to update the current package using a stochastic search algorithm designed for noisy observations, such as adapted versions of simulated annealing [1, 2] and genetic algorithms [30], or other algorithms such as stochastic ruler [3], branch-and bound [21], and so on. The motivation is that such algorithms are designed to minimize the number of scenarios needed and can search though nearby packages rapidly, with no calls to a solver, while also using stochasticity to avoid getting trapped in a local optimum.

Fast sensitivity analysis: Another potentially effective enabler of interactivity is to compute gradient information at the same time that we compute scenarios in order to allow a user to quickly evaluate the effect of small changes in θ. Given a package $\mathcal{P}$, a typical objective function has the form $H_\theta(\mathcal{P}) = \sum_{j\in\mathcal{P}} E_\theta[g(A_j, \theta)]$ for a given random attribute A. For example, A_j might be the random gain for the jth stock, θ might be the interest rate used in the predictive model, and $g(x, \theta)$ might be a function that computes the net present value (NPV) of the gain. The subscript θ on the expectation operator E indicates that the the interest rate affects the predicted NPV of the gain not only explicitly through the function g, but also implicitly by affecting the probability distribution of the random variable A_j. For simplicity, suppose that the random variable A_j has a known probability density function f_θ. Under mild regularity conditions, it is known [14] that $\nabla_\theta E_\theta[g(A_j, \theta)] = E_\theta[Y_j]$, where $Y_j = g'(A_j, \theta) + g(A_j, \theta)L'(0)$, $g' = \partial g/\partial\theta$, and $L(h) = f_{\theta+h}(A_j)/f_\theta(A_j)$. The quantity L is a *likelihood ratio* for A_j. We then find that $\nabla_\theta H_\theta(\mathcal{P}) = \sum_{j\in\mathcal{P}} E_\theta[Y_j]$. Thus when generating a scenario value of A_j, we can simultaneously compute the quantity Y_j. Averaging the Y_j values over all of the scenarios in a large validation set for each $j \in \mathcal{P}$ and then summing over j, we compute$\nabla_\theta H_\theta(\mathcal{P})$ (with negligible error) and we can then estimate the change in the objective value under small perturbations of θ. That is, if θ' is such a perturbation, then we estimate the modified objective value as $H_{\theta'}(\mathcal{P}) \approx H_\theta(\mathcal{P}) + \nabla_\theta H_\theta(\mathcal{P})(\theta' - \theta)$. Importantly, it is shown in [14] that the quantity $L'(0)$ can be computed even when X_j is the output of a complex predictive simulation model so that the density f_θ does not have a known closed form. Moreover, the likelihood can often be computed with little additional coding or computational effort. We postulate that this technique and related gradient-estimation techniques (see, e.g., [22]) can be applied broadly to enable this interactive functionality, and can also be applied to VaR and CVaR constraints in order to quickly assess the degree of constraint violation under small perturbations of θ.

Precomputation techniques: A standard way to facilitate interactivity in an analytics system is to perform expensive computations offline, precomputing various quantities that can then be exploited to speed up real-time interaction. For example, VG functions that are expensive to compute during scenario generation can be barriers to interactive SPQ evaluation. One possibility for ameliorating this problem is to precompute approximations to the underlying distribution sampled by the VG function—such as the histogram, quantile, or kernel-density approximation methods mentioned in Section 3.3 in the context of partitioning for the purpose of sketching—and then sample from the approximate distribution in real time. As before, there is an interesting trade-off between accuracy, storage requirements, and real-time computation speed. Another potentially powerful application of the precomputation principle is *metamodeling*. The idea is to evaluate a carefully chosen test set of SPQ queries offline using various values of, e.g., chance-constraint parameters (thresholds and risk probabilities) as well as parameters of the underlying predictive models, and build an approximate metamodel that maps these features to the optimal value of the objective function. The user can then interactively change these parameters to explore their effect on the objective function, e.g., to see how varying interest rates might affect the expected gain from an optimal portfolio. Such a metamodel can be viewed as a "global" analogue to the "local" derivative estimation idea discussed earlier. Traditional metamodeling methods for stochastic models include regression modeling and Gaussian-process modeling [5]; more recently, neural network models have become increasingly popular [11, 18]. When modifying constraint values, the idea (as with sensitivity analysis) is to quickly estimate whether the benefits of computing a modified package are worth the computational effort. If so, then the SPQ query can be re-computed (possibly incrementally).

5 Policy Making

Autonomous decision-making agents tightly connect the frameworks of prescriptive and predictive analytics, leading to further research challenges. Such agents are often guided by a *policy*: a function that examines the current state of its environment to select an action that leads to the highest possible future rewards. Figure 5 illustrates how reinforcement learning agents can operate in real-world scenarios. Consider an auto-trader, which utilizes a stock time-series predictive model to simulate the effects (rewards) of different purchase decisions. This allows for a *planning* phase where the agent can experiment with different decisions to approximate the *value* of being in different possible states and to learn a policy that allows the agent to take an *action* that leads to high-valued states. In this setup the predictive model is constantly updated from observations of the real world and the agent constantly updates its internal value and policy models to better determine which stocks to buy or sell on a daily basis.

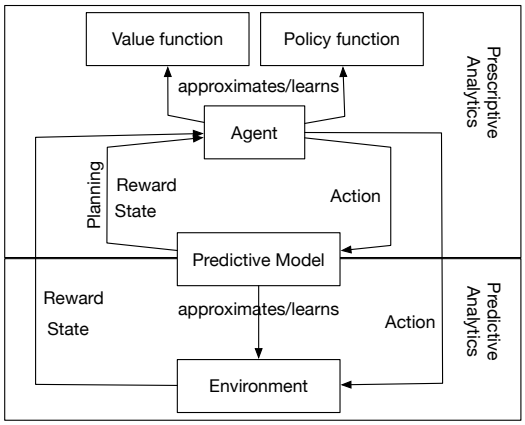


Figure 5: The interplay between predictive and prescriptive analytics in policy-making.

5.1 Scaling the Planning Phase

An auto-trader (the agent) has an astronomical combination of decisions to consider over the planning time horizon. If the auto-trader hopes to maximize profits over a three day time horizon, it explores for each day in that period the predicted price and gain of a certain stock and how many units it should buy or sell. This exploration allows the auto-trader to construct a policy that maximizes expected profit given the current state of stock prices today and expected daily gains. If the stock prediction model updates on a daily basis, planning may need to occur in a relatively short time frame before the stock exchanges open for trading. For even more aggressive hourly traders, planning may need to occur within minutes. State-of-the-art reinforcement learning (RL) techniques such as Soft-Actor Critic (SAC) or Proximal Policy Optimization (PPO) and their variants are computationally intensive involving many simulations to construct policies. This planning phase is a bottleneck to enabling interactive policy making.

ID (i)	stock	day (d)	E[price]	price dist.	E[gain] (g)	gain dist.
1	AAPL	1	150	$f_1(1, ...)$	8	$g_1(1, ...)$
1	AAPL	2	158	$f_1(2, ...)$	12	$g_1(2, ...)$
1	AAPL	3	170	$f_1(3, ...)$	10	$g_1(3, ...)$
2	TSLA	1	758	$f_2(1, ...)$	-4	$g_2(1, ...)$
2	TSLA	2	754	$f_2(2, ...)$	-10	$g_2(2, ...)$
2	TSLA	3	744	$f_2(3, ...)$	-20	$g_2(3, ...)$
			...			

Figure 6: Predicted stock prices over a three-day horizon. VG functions $f_i(d, ...)$, and $g_i(d, ...)$ describe the uncertainty of the price and of the gain of a stock i on day d.

Reformulating Planning as a Stochastic Package Query: One approach is to replace RL methods, such as SAC or PPO, with stochastic package query formulations. In the case of the auto-trader, the predictive model is used to build a relation similar to the one in Figure 6.

The agent can construct a stochastic package query that decides how many stocks to buy or sell (short) given constraints on the principal budget and subsequent estimated daily profits. The SPAQL query is similar to the one introduced in Section 2.2 with a buy/sell decision variable associated with each stock for each day in the planning time horizon and with the following additional constraints:

$$\mathsf{SUM}(\mathsf{price} \times (\mathsf{day} = 2)) \leq 50{,}000 - \mathsf{SUM}(\mathsf{price} \times (\mathsf{day} \leq 1)) + \mathsf{SUM}(\mathsf{gain} \times (\mathsf{day} \leq 1)) \text{ AND}$$
$$\mathsf{SUM}(\mathsf{price} \times (\mathsf{day} = 3)) \leq 50{,}000 - \mathsf{SUM}(\mathsf{price} \times (\mathsf{day} \leq 2)) + \mathsf{SUM}(\mathsf{gain} \times (\mathsf{day} \leq 2))$$

This reformulation allows us to reuse many of the discussed techniques for scaling stochastic packages directly.

5.2 Tracking and Explaining Policy Changes

As the predictive model updates, so does the agent's value and policy models and subsequently the actions taken by the agent. In many situations, a human may wish to interrogate the autonomous agent to understand its reasons for choosing a certain action, especially if the action differs from past actions taken in similar states. To enable this form of *why-so* questioning, we need to build a provenance system that tracks and connects changes across the predictive model and the agent's value and policy models. The ability to answer simple why questions through provenance will form the basic building blocks of interpretability and explainability in prescriptive analytical frameworks.

6 Conclusions

Prescriptive analytics plays a key role in a broad variety of domains, but has received relatively little attention from the database community. As optimization problems become increasingly data-intensive, database researchers are poised to make key contributions to the creation of scalable, seamless, data-centric tools for supporting decision making in complex, uncertain, dynamic environments. Our work on one useful class of optimization problems, package queries, has demonstrated the challenges and opportunities arising from the deep integration of data management, predictive analytics, and optimization software and algorithms. Many challenges remain, both in exploring a broad class of optimization problems arising in applications, and in building systems that enable improved, data-driven decision making.

Acknowledgments. This material is based upon work supported by the ASPIRE Award for Research Excellence (AARE-2020) grant AARE20-307, the NYUAD Center for Interacting Urban Networks (CITIES), and funded by: Tamkeen under the NYUAD Research Institute Award CG001, and the National Science Foundation under grants IIS-1453543 and IIS-1943971. We thank Matteo Brucato, Riddho Haque, Anh Mai, and Nishant Yadav for their many contributions to this work.

References

[1] T. M. Alkhamis and M. A. Ahmed. Simulation-based optimization using simulated annealing with confidence interval. In *Proc. Winter Simulation Conference*, 2004.

[2] M. H. Alrefaei and S. Andradóttir. A simulated annealing algorithm with constant temperature for discrete stochastic optimization. *Management science*, 45(5):748–764, 1999.

[3] M. H. Alrefaei and S. Andradóttir. Discrete stochastic optimization using variants of the stochastic ruler method. *Naval Research Logistics (NRL)*, 52(4):344–360, 2005.

[4] S. Arumugam, R. Jampani, L. Perez, F. Xu, C. Jermaine, and P. J. Haas. MCDB-R: Risk analysis in the database. In *VLDB*, pages 782–793, 2010.

[5] R. R. Barton. Tutorial: Metamodeling for simulation. In *Proc. Winter Simulation Conference (WSC)*, pages 1102–1116, 2020.

[6] M. Brucato, A. Abouzied, and A. Meliou. Package queries: efficient and scalable computation of high-order constraints. *VLDB J.*, 27(5):693–718, 2018.

[7] M. Brucato, J. F. Beltran, A. Abouzied, and A. Meliou. Scalable package queries in relational database systems. *PVLDB*, 9(7):576–587, 2016.

[8] M. Brucato, M. Mannino, A. Abouzied, P. J. Haas, and A. Meliou. sPaQLTooLs: A stochastic package query interface for scalable constrained optimization. *Proc. VLDB Endow.*, 13(12):2881–2884, 2020.

[9] M. Brucato, N. Yadav, A. Abouzied, P. J. Haas, and A. Meliou. Stochastic package queries in probabilistic databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 269–283, 2020.

[10] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. M. Jermaine. Simulation of database-valued Markov chains using SimSQL. In *SIGMOD*, pages 637–648, 2013.

[11] W. Cen and P. J. Haas. Enhanced simulation metamodeling via graph and generative neural networks. In *2022 Winter Simulation Conference*, to appear.

[12] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. *SIGMOD Rec.*, 29(2):391–402, May 2000.

[13] D. Frazzetto, T. Dyhre Nielsen, T. Pedersen, and L. Siksnys. Prescriptive analytics: a survey of emerging trends and technologies. *The VLDB Journal*, May 2019. DOI: 10.1007/s00778-019-00539-y.

[14] P. W. Glynn. Likelihood ratio gradient estimation for stochastic systems. *CACM*, 33(10):75–84, 1990.

[15] P. J. Haas, P. P. Maglio, P. G. Selinger, and W. C. Tan. Data is dead... without what-if models. *PVLDB*, 4(12):1486–1489, 2011.

[16] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. The Monte Carlo Database System: Stochastic analysis close to the data. *ACM Trans. Database Syst.*, 36(3):18:1–18:41, 2011.

[17] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. MCDB: a Monte Carlo approach to managing uncertain data. In *SIGMOD*, pages 687–700, 2008.

[18] H. Lam and H. Zhang. Neural predictive intervals for simulation metamodeling. In *Proc. Winter Simulation Conference (WSC)*, 2021.

[19] I. Lustig, B. Dietrich, C. Johnson, and C. Dziekan. The analytics journey. *Analytics Magazine*, November/December 2010.

[20] A. J. McNeil, R. Frey, and P. Embrechts. *Quantitative Risk Management: Concepts, Techniques and Tools*. Princeton University Press, second edition, 2015.

[21] V. I. Norkin, G. C. Pflug, and A. Ruszczyński. A branch and bound method for stochastic global optimization. *Mathematical programming*, 83(1):425–450, 1998.

[22] Y. Peng, M. C. Fu, J.-Q. Hu, and B. Heidergott. A new unbiased stochastic derivative estimator for discontinuous sample performances with structural parameters. *Operations Research*, 66(2):487–499, 2018.

[23] L. L. Perez, S. Arumugam, and C. M. Jermaine. Evaluation of probabilistic threshold queries in MCDB. In *SIGMOD*, pages 687–698, 2010.

[24] L. Siksnys and T. B. Pedersen. SolveDB: Integrating optimization problem solvers into SQL databases. In *SSDBM*, pages 14:1–14:12, 2016.

[25] L. Siksnys, T. B. Pedersen, T. D. Nielsen, and D. Frazzetto. SolveDB+: Sql-based prescriptive analytics. In *Proc. EDBT*, pages 133–144, 2021.

[26] B. W. Silverman. *Density estimation for statistics and data analysis*. Routledge, 2018.

[27] J. E. Smith and R. L. Winkler. The optimizer's curse: Skepticism and post-decision surprise in decision analysis. *Management Science*, 52(3):311–322, 2006.

[28] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool, 2011.

[29] R. J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer Nature, 2020.

[30] H. Xiao and L. H. Lee. Simulation optimization using genetic algorithms with optimal computing budget allocation. *Simulation*, 90(10):1146–1157, 2014.

User Interfaces for Exploratory Data Analysis: A Survey of Open-Source and Commercial Tools

Jinglin Peng[†*] Weiyuan Wu[†*] Jing Nathan Yan[‡] Danrui Qi[†]
Jeffrey M. Rzeszotarski[‡] Jiannan Wang[†]
[†] Simon Fraser University, Canada {jinglin_peng, youngw, dqi, jnwang}@sfu.ca
[‡] Cornell University, USA {jy858, jeffrz}@cornell.edu

Abstract

Exploratory Data Analysis (EDA) is an important step in data processing pipelines, for which many tools have been developed. In this article we take an initial step in evaluating the relationship between users and EDA tools, as mediated by their interfaces, across different stages of the EDA process. We compare 11 commercial and open-sourced tools, categorizing their interfaces with respect to three stages of EDA: 1) generating initial questions, 2) creating visualizations, and 3) examining visualizations. Finally, we discuss trade-offs between different interfaces, and identify future work and research opportunities for the broader EDA technical community.

1 Introduction

Data processing is the procedure to transform collected raw data into useful information. When processing data in the course of conducting a broader investigation, analysts develop a pipeline or set of activities with linked outputs and inputs, for their work. A typical data processing pipeline includes acquiring data from different sources, wrangling and transforming data into a format ready for exploration, exploring data to examine its quality, discovering insights, building a model for prediction, and finally reporting results [49].

Exploratory data analysis (EDA) is an important step in the data processing pipeline, providing necessary data profiling and insight discovery prior to in-depth analysis [49]. A number of tools have been proposed from both academia and industry to better support end users making sense of data [11,13,23,24,26,31,39,41,42,46,47,50,52]. Prior work in this space has summarized the features of such tools based on their task affordances (e.g., univariate analysis, bivariate analysis, multivariate analysis). Researchers have also adopted a number of lenses for surveying tools. For example, Staniak et al. reviewed existing libraries by tasks that they can support [40], and Ghosh et al. reviewed whether existing pillars can handle the high volume of data through empirically studying 43 academic and 7 commercial tools [16]. Existing work focused on the tasks into which tools can fit, however, risks overlooking or under-emphasizing the human-in-the-loop nature of EDA. The human interface of the tool is indeed the essential connection point between the system and an analyst's data processing pipeline, and merits additional investigation. In this paper, we aim through synthesizing existing pillars of tools, to understand how

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]The first two authors contributed equally to this work.

users make sense of data through different interfaces. As interfaces are closely tied to different EDA stages [9], we contextualize the impact of interfaces from tools in different EDA stages of the data processing pipeline.

One key challenge we face in this work is in aligning the use of tools with stages of the EDA process. Research communities have extensively examined the EDA process, which is highly iterative and interactive, but there is no one dominant framework for the EDA process [9,54]. Rather, our understanding is constantly evolving in concert with tools. Early on, Tukey [27] gives one of the earliest definitions of EDA as the following process: *"1) start with some ideas, 2) iterate between asking a question and creating a design, 3) collect data according to the designed experiment, 4) perform a statistical analysis of data, and 5) produce an answer"*; however, as EDA tasks and the datasets to be explored are becoming more sophisticated, recent research identifies that there might not exist schemata for the stages of EDA which are both universal and specific. For example, Shneiderman uses from the visual information seeking mantra: *"overview first, zoom and filter, then details-on-demand"* [38] instead of characterizing exact and implicit stages, and Heer et al. summarize the visual analysis process as *"an iterative process of view creation, exploration, and refinement"* [18].

In an effort to retain specificity in our survey while also acknowledging the reality that there is no one perfect categorization scheme for EDA, we focus on specific user exploration activities as they relate to the interfaces. In this work, we opt to adapt common definitions from Tukey and adapt them for the modern EDA context. More specifically, we remove the "collect data" step as we consider the scenario of conducting EDA on collected data. Besides, since the recent advancement of surveyed tools are mainly about visualizations, we focus on visualizations in this work. We change "create a design" to "create visualizations", and combine the last two steps as "examine visualizations". Note that there are also other ways to gain insights during EDA, such as executing queries and observing their results. Finally, the EDA stages are: 1) *Generate Initial Questions*: get some initial questions to explore. 2) *Create Visualizations*: create visualizations to answer questions. 3) *Examine Visualizations*: examine the created visualizations to produce answers or get some new questions.

We surveyed 11 commercial and open-sourced EDA tools and categorized the provided interfaces of the tools into three stages as discussed before. The remainder of the paper is organized as follows: In Section 2 we briefly introduce the selected EDA tools. Then we give examples to illustrate the meaning of each EDA stage, and describe how the interfaces in each stage can be used to conduct EDA in Section 3. After that, we categorize interfaces into stages and provide a detailed introduction of each interface in Section 4. Finally, we discuss the trade-offs between different types of interfaces we observed and identify areas for future work in Section 5.

2 EDA Tools

This section presents the EDA tools we examine in this work. We first describe the methodology for our tool selection and then introduce each EDA tool.

2.1 Selection Methodology

The 11 tools we examine fall into two categories based on user interfaces of creating visualization: programming interface tools (Code) and graphical interface tools (No Code). Programming interface tools require a user to write code for creating visualizations, while graphical interface tools provide visualizations through traditional WIMP (windows, icons, menus, pointer) interaction techniques. All investigated programming interface tools are open-sourced, and the graphical interface tools are mixed with open-sourced and commercial tools.

For programming interface tools, we categorize them into task-centric, recommendation-centric, plot-centric, and profiling-centric based on their intended usage (see Section 4.2 for more details). Then we choose one of the most popular Python-based libraries under each usage category. The popularity is judged by Github stars because stars approximate the popularity of an open-sourced library [30]. We focus on Python because it is the most popular programming language in data science, programming language indices [32,44], and an area of high innovation in the EDA space.

For graphical interface tools, we first identify popular commercial options by following a recent strategic report from Gartner Inc. [15] including four categories (Leaders, Challengers, Visionaries, and Niche Players) of BI tools. We choose two tools (Microsoft Power BI [11] and Salesforce Tableau [42]) from the Leaders category, one tool (Google Looker [17]) from the Challengers category, one tool (SAS Viya [35]) from the Visionaries category and one tool (Amazon Quicksight [6]) from the Niche Players category. For open-sourced graphical tools, we include two popular ones, Apache Superset [7] and Voyager2 [50]. Our survey set is listed in Table 7.

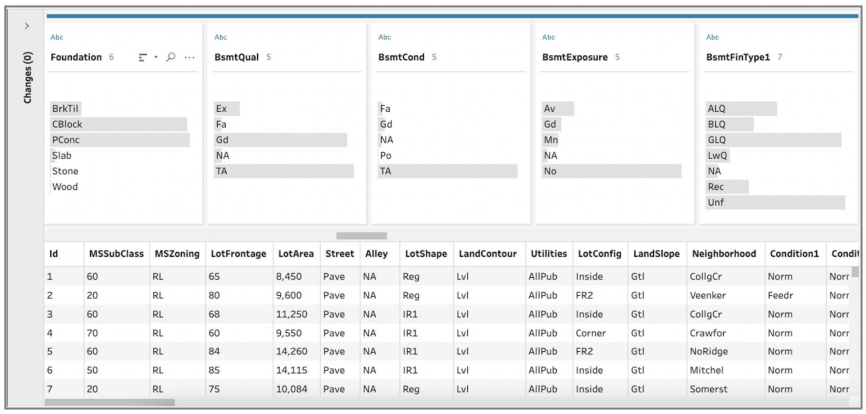
2.2 Programming Interface Tools

Pandas. Pandas [29] is one of the most popular data analysis and manipulation libraries for tabular data in Python. When using Pandas, a user needs to manually load the dataset into Pandas and then generate a visualization by calling the plotting functions provided by Pandas or using a third-party visualization library. This process usually requires certain knowledge of the programming language and the tool itself.
PandasProfiling. PandasProfiling [2] is a profiling-centric tool that can generate a profiling report on a given dataset requiring only one line of code. The profiling report contains visualizations like distributions, correlation matrices, as well as stats and insights based on the dataset.
DataPrep.EDA. DataPrep.EDA [31] is a task-centric EDA library. It provides functions that are named after a specific EDA task, e.g. 'plot_correlation' for doing correlation analysis, for users to complete EDA tasks using one line of code. It supports auto-insight for the produced visualizations.
Lux. Lux [24] is a rec-centric EDA library that automates the visualization process. When using Lux, a user uses the 'intent language' to express their interest in a dataset column and optionally specify rows in the dataset. Lux will then generate a visualization for the interested column and rows, recommending related visualizations.

2.3 Graphical Interface Tools

All the tools in this category do not require a user to write code, but some support using code to access advanced features. The basic workflow for graphical interface tools is the following: 1) import a dataset by uploading files or connecting to a database, 2) specify a visualization type and columns set as the x and y-axis to generate a visualization, and 3) view the results.
Microsoft Power BI. Power BI [11] is a business intelligence tool that allows users to create interactive visualizations of data. Power BI follows a basic office application workflow but can also automatically generate several visualizations after importing the data (without requiring a user to specify the visualization type and the column set). Power BI can also generate insights like trends and anomalies on the whole dataset and on a specific visualization.
Salesforce Tableau. Tableau [42] is a data analytics software and platform. Like Power BI, Tableau also automates visualization creation by specifying the type of visualization and the required data transformation after a user chooses the column set for the x and y-axis. Tableau can also generate visualizations based on natural language queries.
Google Looker. Google Looker [17] is a cloud BI service based on the Google Cloud Platform (GCP). It directly connects to data warehouses and creates visualizations from the query result on the data source. After a visualization is created, Looker can add forecasts into the visualization using machine learning. Looker also supports generating visualizations from natural language queries.
Amazon QuickSight. Like Google Looker, Amazon QuickSight [6] is a cloud BI service based on AWS. QuickSight can suggest insights based on the data automatically. QuickSight also supports generating visualizations using natural language by 'Amazon QuickSight Q'.
SAS Viya. SAS Viya [35] is an AI, analytic and data management platform. Viya follows a standard office tool workflow to generate visualization and can provide auto-insights through the automated analysis feature.

(a) Data profiling page in Tableau Prep Builder



(b) Auto-insight (red box) in PowerBI

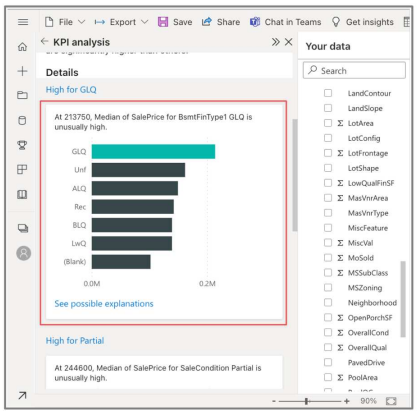


Figure 1: Examples of Generating Initial Questions Using Tableau and PowerBI

Apache Superset. Apache Superset [7] is an open-sourced business intelligence web application. Superset follows the basic workflow to produce visualizations but supports a wide variety of visualizations, including not only common ones like bar charts but also geography, flow diagrams, and more.

Voyager 2. Voyager 2 [50] is an open-sourced visualization tool for data exploration. Voyager 2 supports both manual and automated chart specifications for generating visualizations. That is, after importing a dataset, Voyager 2 will automatically generate several visualizations. After that, the user can specify columns or wildcards (e.g. all categorical columns) to guide Voyager 2 to generate more related visualizations.

3 Examples of Conducting EDA

To let the reader have a better overview of how different types of interfaces can be used throughout an EDA process before we dive into the details, we demonstrate an EDA process using the task of predicting the SalePrice of a home on the publicly available House Prices dataset [10]. The House Prices dataset contains the sale price of residential homes in Ames, Iowa, with 79 explanatory variables. Amongst the 79 explanatory variables, there are 51 categorical variables and 28 numerical. Some of them are of high quality, and some contain many missing values. Additionally, only part of the variables is correlated with the SalePrice.

Generate Initial Questions. An EDA process usually starts with skimming through each column to get an overview of the dataset and to come up with some initial questions. This includes checking dataset-wise statistics, inspecting missing values and examining univariate and multivariate distributions. We use Tableau Prep Builder as an example to show how this can be done. As in Figure 1 (a), the profiling report displays the distribution for each column of the dataset as well as a sample of data at the bottom. The distribution of the column 'BsmtFinType1', which is the 'Rating of basement finished area', shows some variance in the distribution while containing a few missing values (NA). This leads us to come up with the initial question: 'Do homes with different BsmtFinType1 affect the SalePrice?'

On the other hand, EDA tools with auto-insights can suggest potentially useful information related to the target column(s). By selecting the 'SalePrice' column as the target column, Power BI (Figure 1 (b)) automatically computed some insights like 'Average of SalePrice for BsmtFinType1 GLQ is unusually high.', guiding us to explore the relationship between the 'SalePrice' column and the 'BsmtFinType1' column.

Create Visualizations. After getting some initial questions in mind, the next step is to create visualizations to answer the question. We create visualizations to examine the relationship between the 'BsmtFinType1' column and the 'SalePrice' column using Code and No Code interface.

With the No Code interface, like in most commercial tools, a user can drag and drop column names into the

visualization area to express the intention of creating a visualization on the specified column. For example, Figure 2 (a) demonstrates how to use the drag and drop feature to create a visualization between 'BsmtFinType1' and 'SalePrice' using Tableau.

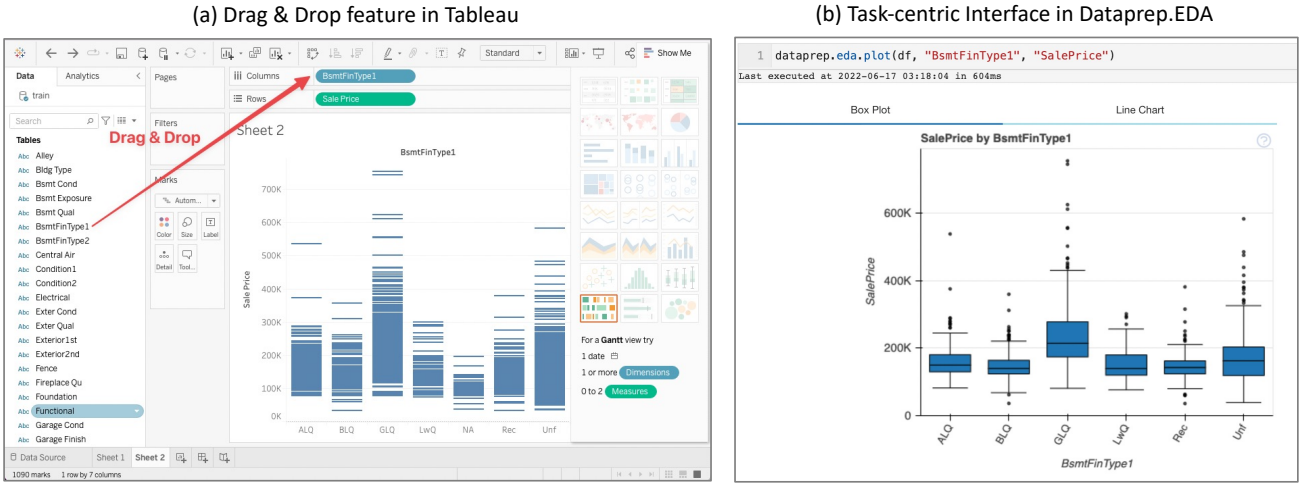


Figure 2: Examples of Creating Visualizations Using Tableau and Dataprep.EDA

On the other hand, a Code interface requires a user to write several lines of code to accomplish the same goal. While this may seem more effortful, for an experienced user or effectively designed API the time required ought to be similar to that of an interactive one. Figure 2 (b) demonstrates how to use the task-centric interface in DataPrep.EDA to create a visualization that helps complete the 'understanding two variables' task.

Examine Visualizations. The Examining Visualizations stage then follows. The first step is directly inspecting the visualization to make sense of its visual scheme, identifying marks and visual features. From Figure 2 (a) we can see that homes with 'GLQ', which stands for 'Good Living Quarters' have a tendency to have a higher price, which agrees with what we get from the auto-insights in Figure 1 (b). Moreover, we can also observe that there are a few extremely high sale price homes, which might be outliers. In this case, we can iterate with the question 'are these high sale price GLQ homes outliers,' and go back to the 'Create Visualizations' step. Such iteration is common in EDA practice.

Other EDA tools also allow users to do different interactions on the visualization. For example, Tableau (Figure 3 (a)) allows us to pick the outliers directly in the visualization and exclude it from the dataset if we do think the extraordinarily high sale price in the 'GLQ' category is an outlier. Power BI can suggest other subgroups that make the distribution different. For example, in Figure 3 (b) Power BI indicates that homes with the 'Foundation' equal to 'BrkTil' (Brick Tile) have very few 'GLQ'. The different ways to examine the visualization provided by the EDA tools help the user to come up with more questions, thus forming a 'Generate Question - Create Visualizations - Examine Visualizations' cycle.

4 Interfaces of EDA Tools

Human interfaces, whether in the form of a programming API or interactive GUI, are an essential component of modern EDA tools. In this section, we outline the commonalities and differences among the surveyed tools and connect them to broader trends in EDA research. We do this with respect to the EDA stages described in the Introduction and outlined in Table 7. We found from existing work and literature a common general pattern that users will go through interactively: generating initial questions, creating visualizations, and examining visualizations for answering initial questions or proposing new questions. We will discuss these stages separately

(a) Manual Interaction (Removing Outliers) in Tableau

(b) Automatic Discovery in PowerBI

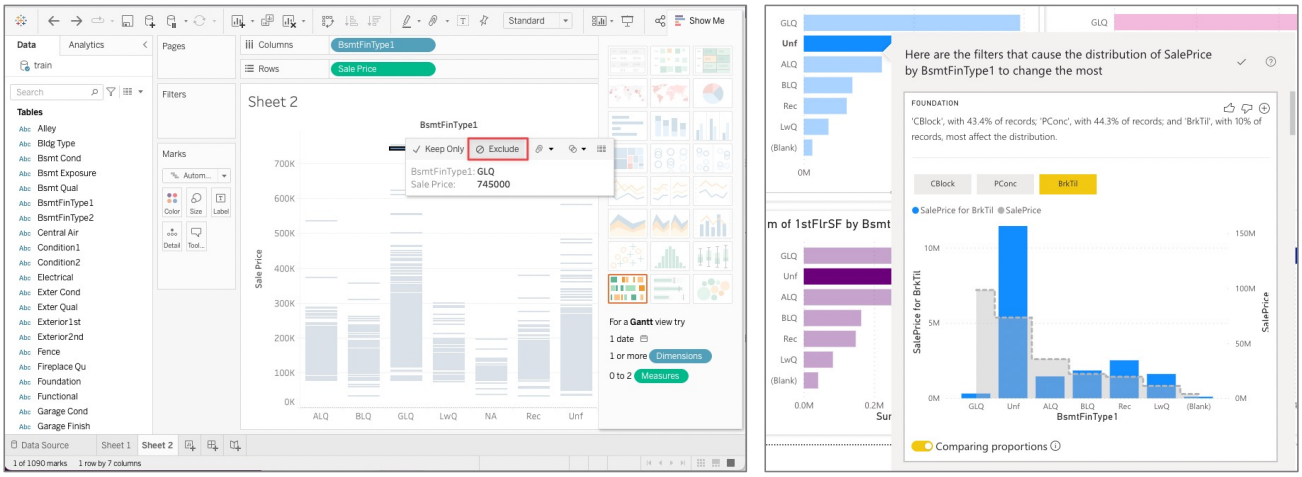


Figure 3: Examples of Examining Visualizations Using Tableau and PowerBI

Table 7: Interface Design of Commercial and Open-sourced EDA Tools

			Tableau	Power BI	Viya	Looker	Quicksight	Superset	Voyage2	Pandas	PandasPorfiling	DataPrep.EDA	Lux
Generate Initial Questions	Data Profiling	Dataset Statistics	✓	✓	✓					✓	✓	✓	
		Univariate Distributions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		Multivariate Distributions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		Missing Value Overview	✓	✓	✓					✓	✓	✓	
	Auto-insight	Distribution	✓	✓	✓		✓				✓	✓	
		Anomalies	✓	✓			✓						
		Trend		✓									
Create Visualizations	No Code	Template	✓	✓	✓	✓	✓	✓	✓				
		Drag & Drop	✓	✓	✓		✓	✓	✓				
		Natural Language	✓	✓			✓						
	Code	Plot-centric	✓	✓	✓	✓	✓	✓		✓			
		Task-centric										✓	
		Profiling-centric									✓		
		Rec-centric							✓				✓
Examine Visualizations	Manual Interaction	Select	✓	✓	✓	✓	✓	✓					✓
		Explore	✓	✓	✓	✓	✓	✓	✓				
		Reconfigure	✓	✓		✓	✓	✓	✓				
		Encode	✓	✓		✓	✓	✓	✓				
		Abstract/Elaborate	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
		Filter	✓	✓	✓	✓	✓	✓	✓				
		Connect	✓	✓		✓	✓	✓					
	Automatic Discovery	Auto-explanation	✓	✓	✓								
		Auto-insight	✓	✓			✓						
		Recommendation	✓	✓	✓				✓				✓

in the following subsections.

4.1 Generate Initial Questions

The first stage we will discuss concerns the generation of initial insights and hypotheses during EDA. As EDA's primary aim is to expose data through open-ended exploration [45], the first step of EDA is often to generate initial questions as starting points. We observed two approaches in our set of tools to generate the initial questions: data profiling and auto-insight.

Data Profiling. Traditionally, the starting point of exploratory data analysis is *"overview first"* [38], which can be achieved by creating a data profiling report using tools like Pandas-Profiling [2] and DataPrep.EDA [31]. There are several aspects of overview information in data profiling which we observe among our surveyed tools. The first aspect is providing summary statistics depicting the basic information of the dataset, such as the number of rows, the number of columns, the number of columns that contain missing values, and so on. The second aspect is providing distributional information like univariate distributions which depict the distribution of each column and multi-variate distributions which show the union distribution of multiple columns. And the final aspect is

providing an overview for the distribution of missing values, such as the missing value percentage of each column and the missing value location distribution of each column.

Data profiling often involves many potential avenues of exploration (and therefore tool affordances). Some of them may be useful and some of them may be redundant information. Thus, data profiling requires a degree of expertise if the interface does not provide obvious suggestions and hooks for the next steps. As many non-experts also need tools for profiling, there is great potential for future tools in automatically filtering useful information and providing inspiration. Auto-insight may be suitable for inspiring these non-experts for asking some questions with possible insights.

Auto Insight. While the profiling report provides a way for users to observe data and come up with some questions, commercial tools like PowerBI and open-source libraries like Dataprep.EDA provides another approach to start the exploration, automatically generating insights [14], which may be more convenient for non-technical or time-sensitive users. The insights are detected by some predefined rules. If a visualization satisfies a rule, then an insight is reported. We identify three types of insights among our set of tools. The first type is detecting interesting distributions like highly skewed distributions or low-variance numerical columns. Also, detecting anomalies like outliers out of 3σ range is also an important type. The last type is detecting interesting trends in time series like periodic patterns or significant changes.

4.2 Create Visualizations

After users have some questions in mind, the most common next stage in an EDA process is to investigate these questions more deeply. To accomplish this goal, the tools we examined offer means to create visualizations to probe the data about user questions and hypotheses. The means of creating visualizations can be divided into two types based on whether users need to write code: *No Code* and *Code*.

No Code. The *No Code* approaches provide a convenient way for non-technical users that are not familiar with programming languages to create visualizations. The first way is providing templates for users to fill out. According to the user-provided information like selected columns and visualization types in templates, the visualizations are automatically created. The second way is providing users with an interface to select the visible objects of interest [48] like the columns and visualization types by dragging that used to create the visualization. If users want to change the selected visible objects, just drop them and switch to new ones. These visible objects can significantly save users' time by avoiding writing complex declarative languages or code. The third way is providing users natural language interface [36] including natural language text and speech [43], which allows users directly input a question using natural language and then recommend the most related visualizations to answer this question. The learning curve for these tools is thus much reduced, assuming recommendations are of high accuracy and users can adequately describe their interests [37].

Code. While *No Code* approaches may be more convenient for non-technical users, their interfaces may be less flexible for advanced users. For this reason, EDA tools also often offer *Code* interfaces which provide analysts with more complex and adjustable tools (albeit with higher complexity and training requirements). Based on the granularity of the API, the *Code* interface has four common types that we have observed: *Plotting-Centric*, *Recommendation-Centric*, *Profiling-Centric* and *Task-Centric*. *Plot-Centric* interfaces provide APIs which are used to create a specific type of visualization. For example, the `plot.hist` in Pandas is used to create histograms, and the `plot.bar` is used to create bar charts. *Recommendation-Centric* interfaces provide APIs which can give users recommendations according to the input datasets. However, the recommendation strategies are often in one manner. For example, Lux [24] automatically recommends bivariate correlation, univariate distribution and occurrence. *Profiling-Centric* interfaces provide APIs which can produce multiple visualizations with one manner for all datasets. *Task-Centric* interfaces provide declarative APIs which are create all the visualizations required by a specific EDA task. For example, DataPrep.EDA [31] provides APIs for statistical modeling tasks. If users want to create visualizations about missing values, just call `plot_missing` in Dataprep.EDA. Compared to plotting-centric interfaces which are under a grain granularity and profiling-centric interfaces which are under

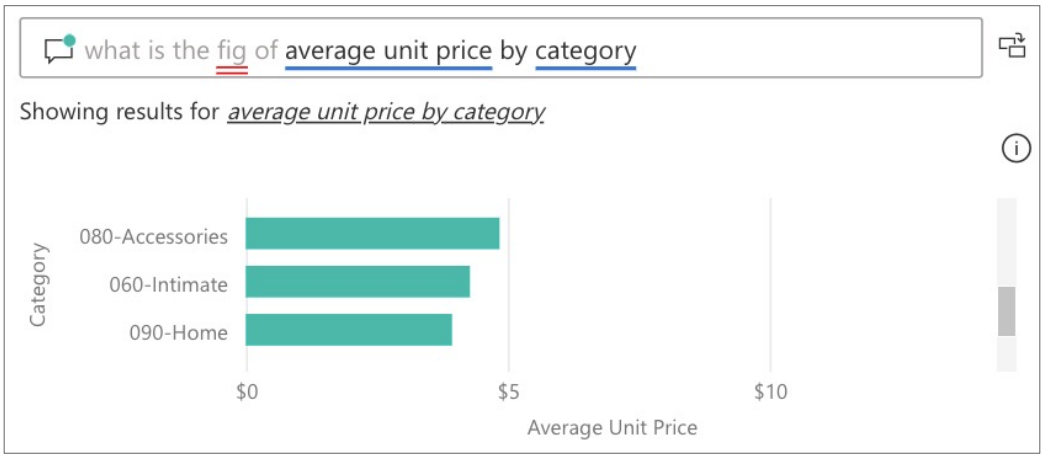


Figure 4: An example of the asking question feature (the Q&A in Power BI)

a coarse granularity, task-centric interfaces leverage a trade-off of visualization granularity. Also, task-centric interfaces are easy to customize, e.g. adjusting bin sizes when plotting bar charts for a specific EDA task.

4.3 Examine Visualizations

After visualizations are created, the last step is to examine the visualizations. The goal is to answer initial questions and find something interesting for proposing new questions. To examine the visualizations, one can manually interact with the created visualizations to develop an understanding. The system itself can also examine visualizations by automatically discovering possible insights and explanations, and providing recommendations. Hence, we classify the interface of examining visualizations into two categories: *manual interaction* and *automatic discovery*.

Examination with Manual Interaction. There exist many works that summarize the interactions for examining visualizations from different perspectives. We leverage the classification in a survey [55]. As shown in Table 7 (more specifically, *Examine Visualizations — Manual Interaction*), it classifies the interactions into seven types. Users often choose the *select* interaction to highlight their most interesting items in created visualizations. The way of highlighting can be a color change, or an increase of the shape size. In the other way, users can employ the *explore* interaction to explore a different collection of items. One example is panning the map. The *reconfigure* interaction allows users to change the *"spatial arrangement of representations"* [55] in a visualization, e.g. when users want to sort bar charts according to the heights of bars, the sorting operation rearranges the bars. The *encode* interaction can change the representation of the visualizations, such as changing the bar chart to pie chart, and changing the color and shape of each item. The *Abstract/Elaborate* interaction allows the user to see more or fewer details, such as zooming-in for more details and zooming-out for less details (more overviews). The *filter* interaction can change the data used for visualization by only keeping items satisfying some conditions. For example, if the users are interested in the subset of data with condition `age < 20`, they can add a filter `age < 20` to visualize this subset. The *connect* interaction is used to highlight corresponding parts among multiple visualizations for selected items. When different visualizations are created for the same dataset, the user can select one item from one visualization, and the *connect* interaction can highlight the corresponding parts related to the selected item in other visualizations.

Examination with Automatic Discovery. Apart from manual interaction to examine the visualizations, the system can automatically discover interesting insights and possible explanations to some questions, which helps users to examine whether the created visualizations can answer their initial questions. Three ways of automatic discovery are summarized from our set of tools. The first way is also called *auto-insight*. Compared to the

auto-insight mentioned in Section 4.1 which detects the insights over the full data, the *auto-insight* here just focuses on the data leveraged by created visualizations, i.e. subsets of data. These insights may help users to answer initial questions and propose new questions. The second way is called *auto-explanation*, which can automatically generate possible explanations for interesting aspects in created visualizations. These explanations may answer initial questions and inspire further questions. The third way is called *recommendation*. Apart from the created visualizations, some systems can also recommend related visualizations by changing elements in current visualizations, which may be helpful to find the answer for initial questions or generate further questions. For example, Lux [24] can automatically recommend related visualizations by making some changes to existing visualizations (e.g., add an additional attribute, add a filter or remove an attribute), which provides users answers and inspirations with external information.

5 Discussion

In this section, we discuss the current status of interfaces for commercial and open-source EDA tools on different stages, and identify possible research opportunities.

5.1 Generate Initial Questions

Profiling for dependency relationships. We observed most tools only provide profiling for basic statistics of a single column and at most two columns. However, it lacks the support of more complex profiling such as inclusion dependency and functional dependency, which are widely studied in the database community. This limits the insights that users can observe from the profiling result. The challenge here is how to support more complex profiling and make the process efficient. For example, some approximate approaches can be used. For more types of profiling and more discussions on this topic, we refer interested readers to a data profiling survey [4].

From descriptive insights to prescriptive insights. We observed the insights provided by most tools are descriptive (e.g., a group-by query has a very large group), while the research community has many studies on prescriptive insights [8, 25, 51] (e.g., explaining why the group size is too large). We have seen the trend of supporting these prescriptive insights in EDA tools. For example, Tableau's "Explain Data" feature gives possible explanations for unexpected aggregate values, such as "why is the average housing price too high?".

Seamlessly integrate profiling and auto-insight. Profiling and auto-insight have their own advantages and limitations. For profiling, it creates many visualizations that cover the basic data characteristics of all columns. However, users may need to check a lot of visualizations until they find something interesting. For auto-insight, it directly reports the visualizations that users may be interested in, thus they can spend less time finding something interesting. However, it may not align with users' real interests or cover all the columns (e.g., users are interested in column A, but none of the returned insights contains column A).

Given the pros and cons of profiling and auto-insight interfaces, one natural question is whether we can combine them to better serve users. For example, if the profiling report contains too many visualizations, then the auto-insight feature can help highlight possible interesting visualizations and prioritize them to users. Furthermore, one can also add an auto-insight section in the report, helping users automatically discover questions that are not covered by observing the report. For this approach, the auto-insight and report should be treated as a whole and the search space of insight should be reconsidered, e.g., by excluding the insight that can be found using other visualizations in the report.

5.2 Create Visualizations

Combining Code and No Code interfaces. The No Code interface is easy to use and has fewer requirements (no programming skill is needed). However, it lacks transparency and users have limited control of the creation

process. On the other hand, the Code interface provides a more flexible way to create visualizations, but it is not accessible to non-programmers.

To make the Code interface easier to use, we observed that there is a trend of adding GUI and interaction in the coding environment like Jupyter Notebook [5, 22]. For example, DataPrep.EDA [31] and Lux [24] generate interactive visualizations, and Bamboolib [1], PandasGUI [3] and Mage [21] provide GUIs for data exploration. By integrating Code with Notebook GUI, the Code interface is not only flexible but also easy to use. To increase the transparency of the No Code interface, one idea is to automatically generate the code for each visualization such that users can copy and paste the code to a programming environment and reproduce the visualization. In this way, users have more flexibility to customize visualizations using a programming language.

Natural language as a unified interface. The No Code interface such as drag & drop is convenient to create visualizations, especially for non-technical users. The advanced feature such as supporting natural language interfaces takes one step further. It allows users who are not even familiar with BI tools to create visualizations. The current support of natural language interfaces is still limited. As shown in Figure 4, the supported questions are relatively simple and follow certain patterns. If a question is not supported, it may recommend the closest question to the user's input. For example, the gray keywords are not understood by the system, thus it suggests a close question and shows its result.

Although natural language support is at an early stage, it has the potential to become a unified interface for EDA. For example, users may first ask "what's interesting about data" to discover insights and get some initial questions. Then the system returns an insight that the average salary in Beijing is much lower than in other cities. After that, the user asks "why the average salary is low?" through the natural language interface. Finally the system receives the question and gives an explanation. There are many challenges that exist to build such a system. For example, handling the ambiguity of natural language queries, mapping the query to visual representation and managing the dialogue system [37]. We refer interested readers to a recent survey for more discussion [37].

5.3 Examine Visualizations

Augmented Analytics. One big trend of data analysis is the engagement of artificial intelligence (AI) techniques. Gartner uses the term "augmented analytics" to represent the approach *"that automates insights using machine learning and natural-language generation"*, and shows that it *"marks the next wave of disruption in the data and analytics market"* [34]. As discussed in Section 4, the AI techniques can help at each stage, such as automatically recommending insights to start exploration, providing a natural language interface for creating visualizations, and automatically discovering interesting insights and giving possible explanations to understand visualizations. Although AI techniques are powerful, there still exist some challenges when applying them in EDA scenarios. For example, EDA is a time-sensitive process, thus the efficiency of applying AI techniques could be a concern, especially for a large dataset. Then the question is how to make AI techniques efficient for this situation. There exist some opportunities when we consider the property of EDA scenarios. For example, EDA is a human-in-the-loop process, and AI techniques should be applied to better serve humans. Hence, some decisions can be proposed to humans (e.g., selecting the analyzed variables), rather than computing all possibilities. This can help reduce computing time and improve user experience. Besides, when users make a decision, the thinking time can also be leveraged for computation [12].

Apart from the efficiency issues, there are some other practical issues of AI models [20, 33]. For example, the model is trained on training data, but the data itself may be biased. In this case, the model may guide users in a wrong way, and finally lead to a wrong conclusion. For example, Amazon previously used the recruitment system to decide whether to recruit a candidate or not, which proves to have considered some sensitive factors [53]. For more discussion about how AI can help automate the EDA system, we refer interested readers to a recent survey [28].

Leverage context information. While the discussed systems can automatically discover insights and explanations, the context information is not well leveraged. In the following, we discuss how two types of context, data context

and session context, may help improve the automatic discovery process.

Data Context. The data itself has semantic meanings and is helpful for multiple tasks, such as data integration, data discovery, and automatic data preparation [19]. However, we found this is not well used in the studied tools. For example, one type of auto-insight is to aggregate numerical columns in each group and see whether one group has a much larger size than the others. In practice, some aggregations may be meaningless, such as computing the sum of age. If data context is considered, this kind of insight will not be shown to confuse users.

Session Context. As EDA is an iterative process, some discoveries from users might be reused, which can reduce redundant steps in EDA. For example, if an insight is already reported in the last few steps, then it can be regarded as a known fact to users and its weight for ranking can be reduced. Besides, the computation from previous steps may be reused, such that the efficiency can be improved [12].

References

[1] Bamboolib. https://bamboolib.8080labs.com/. Date accessed: 2022-06-07.

[2] Pandas-profiling. https://github.com/ydataai/pandas-profiling. Date accessed: 2022-06-07.

[3] Pandasgui. https://github.com/adamerose/PandasGUI. Date accessed: 2022-06-07.

[4] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *VLDB J.*, 24(4):557–581, 2015.

[5] S. Alspaugh, N. Zokaei, A. Liu, C. Jin, and M. A. Hearst. Futzing and moseying: Interviews with professional data analysts on exploration practices. *IEEE Trans. Vis. Comput. Graph.*, 25(1):22–31, 2019.

[6] Amazon Web Services, Inc. Amazon quicksight: The most popular cloud-native, serverless bi service. https://aws.amazon.com/quicksight. Date accessed: 2022-06-20.

[7] Apache Software Foundation. Apache superset: A modern, enterprise-ready business intelligence web application. https://github.com/apache/superset. Date accessed: 2022-06-20.

[8] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. Macrobase: Prioritizing attention in fast data. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 541–556. ACM, 2017.

[9] L. Battle and J. Heer. Characterizing exploratory visual analysis: A literature review and evaluation of analytic provenance in tableau. *Comput. Graph. Forum*, 38(3):145–159, 2019.

[10] D. D. Cock. Ames, iowa: Alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3):null, 2011.

[11] M. Corporation. Microsoft power bi. https://powerbi.microsoft.com/. Date accessed: 2022-06-20.

[12] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. The case for interactive data exploration accelerators (ideas). In C. Binnig, A. D. Fekete, and A. Nandi, editors, *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, page 11. ACM, 2016.

[13] Z. Cui, S. K. Badam, M. A. Yalçin, and N. Elmqvist. Datasite: Proactive visual data exploration with computation of insight-based recommendations. *Inf. Vis.*, 18(2), 2019.

[14] R. Ding, S. Han, Y. Xu, H. Zhang, and D. Zhang. Quickinsights: Quick and automatic discovery of insights from multi-dimensional data. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 317–332. ACM, 2019.

[15] Gartner Inc. Gartner magic quadrant for analytics and business intelligence platforms. https://www.gartner.com/en/documents/3996944. Date accessed: 2022-06-07.

[16] A. Ghosh, M. Nashaat, J. Miller, S. Quader, and C. Marston. A comprehensive review of tools for exploratory analysis of tabular industrial datasets. *Vis. Informatics*, 2(4):235–253, 2018.

[17] Google LLC. Looker & google cloud. https://www.looker.com/google-cloud/. Date accessed: 2022-06-20.

[18] J. Heer and B. Shneiderman. Interactive dynamics for visual analysis. *Commun. ACM*, 55(4):45–54, 2012.

[19] M. Hulsebos, S. Gathani, J. Gale, I. Dillig, P. Groth, and Ç. Demiralp. Making table understanding work in practice. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022.

[20] H. Kaur, H. Nori, S. Jenkins, R. Caruana, H. Wallach, and J. Wortman Vaughan. Interpreting interpretability: understanding data scientists' use of interpretability tools for machine learning. In *Proceedings of the 2020 CHI conference on human factors in computing systems*, pages 1–14, 2020.

[21] M. B. Kery, D. Ren, F. Hohman, D. Moritz, K. Wongsuphasawat, and K. Patel. mage: Fluid moves between code and graphical work in computational notebooks. In S. T. Iqbal, K. E. MacLean, F. Chevalier, and S. Mueller, editors, *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*, pages 140–151. ACM, 2020.

[22] M. B. Kery, D. Ren, K. Wongsuphasawat, F. Hohman, and K. Patel. The future of notebook programming is fluid. In R. Bernhaupt, F. F. Mueller, D. Verweij, J. Andres, J. McGrenere, A. Cockburn, I. Avellino, A. Goguey, P. Bjøn, S. Zhao, B. P. Samson, and R. Kocielnik, editors, *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems, CHI 2020, Honolulu, HI, USA, April 25-30, 2020*, pages 1–8. ACM, 2020.

[23] T. Kraska. Northstar: An interactive data science system. *Proc. VLDB Endow.*, 11(12):2150–2164, 2018.

[24] D. J. L. Lee, D. Tang, K. Agarwal, T. Boonmark, C. Chen, J. Kang, U. Mukhopadhyay, J. Song, M. Yong, M. A. Hearst, and A. G. Parameswaran. Lux: Always-on visualization recommendations for exploratory dataframe workflows. *Proc. VLDB Endow.*, 15(3):727–738, 2021.

[25] B. Lockhart, J. Peng, W. Wu, J. Wang, and E. Wu. Explaining inference queries with bayesian optimization. *CoRR*, abs/2102.05308, 2021.

[26] Y. Luo, X. Qin, N. Tang, and G. Li. Deepeye: Towards automatic data visualization. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 101–112. IEEE Computer Society, 2018.

[27] W. L. Martinez, A. R. Martinez, and J. L. Solka. *Exploratory data analysis with MATLAB®*. Chapman and Hall/CRC, 2017.

[28] T. Milo and A. Somech. Automating exploratory data analysis via machine learning: An overview. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2617–2622. ACM, 2020.

[29] T. pandas development team. pandas-dev/pandas: Pandas. Feb. 2020.

[30] M. Papamichail, T. G. Diamantopoulos, and A. L. Symeonidis. User-perceived source code quality estimation based on static analysis metrics. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Vienna, Austria, August 1-3, 2016*, pages 100–107. IEEE, 2016.

[31] J. Peng, W. Wu, B. Lockhart, S. Bian, J. N. Yan, L. Xu, Z. Chi, J. M. Rzeszotarski, and J. Wang. Dataprep.eda: Task-centric exploratory data analysis for statistical modeling in python. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2271–2280. ACM, 2021.

[32] G. Piatetsky. Python leads the 11 top data science, machine learning platforms: Trends and analysis. https://www.kdnuggets.com/2019/05/poll-top-data-science-machine-learning-platforms.html. Date accessed: 2022-06-07.

[33] F. Poursabzi-Sangdeh, D. G. Goldstein, J. M. Hofman, J. W. Wortman Vaughan, and H. Wallach. Manipulating and measuring model interpretability. In *Proceedings of the 2021 CHI conference on human factors in computing systems*, pages 1–52, 2021.

[34] R. Sallam, C. Howson, and C. J. Idoine. Augmented analytics is the future of data and analytics. *Gartner, Inc*, 27, 2017.

[35] SAS Institute Inc. Sas viya: Confident decisions at every moment. https://www.sas.com/en_ca/software/viya.html. Date accessed: 2022-06-20.

[36] V. Setlur, S. E. Battersby, M. Tory, R. Gossweiler, and A. X. Chang. Eviza: A natural language interface for visual analysis. In J. Rekimoto, T. Igarashi, J. O. Wobbrock, and D. Avrahami, editors, *Proceedings of the 29th Annual Symposium on User Interface Software and Technology, UIST 2016, Tokyo, Japan, October 16-19, 2016*, pages 365–377. ACM, 2016.

[37] L. Shen, E. Shen, Y. Luo, X. Yang, X. Hu, X. Zhang, Z. Tai, and J. Wang. Towards natural language interfaces for data visualization: A survey. *arXiv preprint arXiv:2109.03506*, 2021.

[38] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, USA, September 3-6, 1996*, pages 336–343. IEEE Computer Society, 1996.

[39] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. G. Parameswaran. Effortless data exploration with zenvisage: An expressive and interactive visual analytics system. *Proc. VLDB Endow.*, 10(4):457–468, 2016.

[40] M. Staniak and P. Biecek. The landscape of R packages for automated exploratory data analysis. *R J.*, 11(2):347, 2019.

[41] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Vis. Comput. Graph.*, 8(1):52–65, 2002.

[42] TABLEAU SOFTWARE, LLC. Tableau. https://www.tableau.com/. Date accessed: 2022-06-20.

[43] J. Tang, Y. Luo, M. Ouzzani, G. Li, and H. Chen. Sevi: Speech-to-visualization through neural machine translation. In Z. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 2353–2356. ACM, 2022.

[44] TIOBE Software BV. Tiobe index. https://www.tiobe.com/tiobe-index/. Date accessed: 2022-06-07.

[45] J. W. Tukey et al. *Exploratory data analysis*, volume 2. Reading, MA, 1977.

[46] J. VanderPlas, B. E. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert. Altair: Interactive statistical visualizations for python. *J. Open Source Softw.*, 3(32):1057, 2018.

[47] M. Vartak, S. Rahman, S. Madden, A. G. Parameswaran, and N. Polyzotis. SEEDB: efficient data-driven visualization recommendations to support visual analytics. *Proc. VLDB Endow.*, 8(13):2182–2193, 2015.

[48] A. Whilden, D. Karis, V. Setlur, R. Degtyar, J. Que, and F. Lymperopoulos. Blocks: Creating rich tables with drag-and-drop interaction. 2022.

[49] K. Wongsuphasawat, Y. Liu, and J. Heer. Goals, process, and challenges of exploratory data analysis: An interview study. *CoRR*, abs/1911.00568, 2019.

[50] K. Wongsuphasawat, Z. Qu, D. Moritz, R. Chang, F. Ouk, A. Anand, J. D. Mackinlay, B. Howe, and J. Heer. Voyager 2: Augmenting visual analysis with partial view specifications. In G. Mark, S. R. Fussell, C. Lampe, m. c. schraefel, J. P. Hourcade, C. Appert, and D. Wigdor, editors, *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017*, pages 2648–2659. ACM, 2017.

[51] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *Proc. VLDB Endow.*, 6(8):553–564, 2013.

[52] E. Wu, F. Psallidas, Z. Miao, H. Zhang, and L. Rettig. Combining design and performance in a data visualization management system. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[53] J. N. Yan, Z. Gu, H. Lin, and J. M. Rzeszotarski. Silva: Interactively assessing machine learning fairness using causality. In *Proceedings of the 2020 CHI conference on human factors in computing systems*, pages 1–13, 2020.

[54] J. N. Yan, Z. Gu, and J. M. Rzeszotarski. Tessera: Discretizing data analysis workflows on a task level. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021.

[55] J. S. Yi, Y. ah Kang, J. T. Stasko, and J. A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1224–1231, 2007.

The Right Tool for the Job: Data-Centric Workflows in Vizier

Oliver Kennedy,[⋆] Boris Glavic,[◇] Juliana Freire,[†] and Michael Brachmann [⋆]
[⋆] University at Buffalo, USA
[◇] Illinois Institute of Technology, USA
[†] New York University, USA

Abstract

Data scientists use a wide variety of systems with a wide variety of user interfaces such as spreadsheets and notebooks for their data exploration, discovery, preprocessing, and analysis tasks. While this wide selection of tools offers data scientists the freedom to pick the right tool for each task, each of these tools has limitations (e.g., the lack of reproducibility of notebooks), data needs to be translated between tool-specific formats, and common functionality such as versioning, provenance, and dealing with data errors often has to be implemented for each system. We argue that rather than alternating between task-specific tools, a superior approach is to build multiple user-interfaces on top of a single incremental workflow / dataflow platform with built-in support for versioning, provenance, error & tracking, and data cleaning. We discuss Vizier, a notebook system that implements this approach, introduce the challenges that arose in building such a system, and highlight how our work on Vizier lead to novel research in uncertain data management and incremental execution of workflows.

1 Introduction

Interactions with data, whether by experts or less technical users, are frequently iterative. As the user explores and transforms her data, she regularly backtracks to correct mistakes, extend her work, or to update source data as it evolves. Throughout this process, users employ a variety of tools, each providing a unique *data interaction modality*. For example, it is common for a data-centric workflow to involve some combination of a web browser (data discovery), a spreadsheet (quick statistics, minor data corrections, data entry), a database (batch manipulation and aggregation over large tabular data), a computational notebook (data visualization and model-building), a scripting IDE (modular component development), command-line tools (quick curation or summary tasks on simple data), and/or a business intelligence dashboard (data visualization).

Each tool provides a unique interaction modality designed for specific tasks and skill levels, but less effective at others. Thus, users working with data frequently string together workflows out of multiple tools, taking the most convenient for each task at hand. Such manual orchestration incentivizes bad habits that create challenges for reproducibility. It is also labor-intensive and error-prone because users have to translate data between the different formats expected by the tools they use, and must manually track provenance and manage documentation. Manual orchestration of data-centric workflows creates collections of workflow artifacts (e.g., datasets, code, and images) based on which it can be hard to (i) debug how a particular unexpected result was reached, or (ii)

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

reapply the workflow when source data changes or if an error has been discovered in the workflow and was fixed by modifying the workflow. Automating such features for workflows that span multiple modalities and systems is a challenging problem that typically entails building an overarching system for the orchestration of workflows and requires changes to the individual systems used in a workflow.

In this article, we outline the design of Vizier[1], an extensible, multi-modal platform for data-centric workflows. At the core of Vizier is a computational notebook, analogous to Jupyter or Zeppelin, that serves to orchestrate workflows. Vizier distinguishes itself from these tools in two ways. First, Vizier's modular architecture allows different interaction modalities to be built as views over the workflow encoded in the notebook. This includes simple views over individual cells or data artifacts that expose data interaction modalities like spreadsheets, as well as more complex modalities for data debugging, data documentation, and more. Second, Vizier's feature-rich workflow engine supports automatic refresh of stale outputs, fine-grained provenance features like propagation of information about uncertainty in data and data documentation through workflow steps, and supports workflows encoding notebooks where the dataflow between cells is only learned at runtime.

Vizier provides a simple workflow abstraction over which its extensible views are built: a workflow is a linear sequence of steps (called *cells* by analog to computational notebooks) and each step manipulates named, typed *artifacts*. Crucially, in Vizier, cells are isolated from each other and communicate only through a shared API. All inter-cell interaction takes place through cells reading and writing artifacts. This is in contrast to notebooks like Jupyter, where inter-cell interactions occur through the global state of an interpreter for a scripting language like Python. Vizier provides a streamlined mechanism for ensuring that workflow state (the versions of artifacts produced by the workflow) is re-computed when necessary. Each cell sees exactly those artifact versions that would be produced by executing all preceding cells in the order they appear in the notebook. The Vizier workflow engine also has built-in support for versioning workflows and artifacts. Every edit to a cell results in a new workflow version being created and associated with the corresponding versions of the artifacts it reads and creates. Both cells and artifacts are (conceptually) immutable and, as in functional data structures, an artifact or cell version can be shared by multiple versions of a workflow.

In what follows, we describe the Vizier workflow engine, as well as several views built on-top of this engine, including Vizier's notebook interface, a spreadsheet mode for interacting with datasets, a history viewer, and a summary view for data errors. We start with a motivating story of a user interacting with data.

1.1 User Story

Alice is organizing several hundred student volunteers for a multi-day online event. She asks volunteers to complete an online form with details including name, institutional email address, and time zone. At some point, many, although not all volunteers have completed the form and she is able to access the results as a downloadable CSV file. The event organizer has asked for a plot summarizing volunteer availability at different times of day, based on each participant's time zone. The availability window for students in adjacent time zones overlaps, making this a challenging problem for spreadsheets, but trivially solvable in SQL (given a table of time zone to availability mappings). Alice downloads the CSV file, ingests it into a relational database, runs the query, exports the result, and imports it into a spreadsheet to create a plot. It is convenient for Alice to use a mix of tools, because each tool is specialized for a particular subtask. However, she now has to manually manage multiple versions of her dataset across the tools.

> **Takeaway:** Data science benefits from chaining together multiple data interaction modalities, including but not limited to Exploration/Discovery, Direct Manipulation, and Querying/Scripting.

She then discovers that some volunteers have submitted multiple responses via the form. Automated curation libraries or stand-alone curation tools [22, 24] are usually sufficient to find and repair duplicate keys. However,

[1] `https://vizierdb.info`

the tool Alice selects is not able to handle several responses where both the original and revised form submissions contain distinct typos, a fact she does not realize until later.

> **Takeaway:** Many data curation tasks like key repair can be automated, but uncertainty arising from heuristics needs to be tracked and communicated to users [25, 37].

With each error corrected, Alice must return to the workflow and repeat all of the steps she previously performed so that she can generate a new plot that reflects these changes.

> **Takeaway:** Data workflows are developed iteratively; revisions to a prior step may require re-execution of subsequent steps to ensure that the results produced by these steps is up-to-date.

Alice's next task is to generate a credits page for the event's website, but she realizes that she has not asked the students to provide a home institution. Fortunately, using a Python script she can heuristically fetch institution names based on the domain name of the students' email address, by scraping the institution's website and retrieving the organization name from the web page metadata. Unfortunately, the script fails on a small number of unusual email addresses — for example some institutions have a separate root domain name for student emails. Writing a script that fixes each outlier individually would be more effort than it is worth, so Alice opens the script output in a spreadsheet and manually enters institutions for the missing students.

> **Takeaway:** The 80/20 rule applies to data science: uncommon error are often easier to fix manually.

She then feeds the resulting spreadsheet into a new script that generates a credits page. As the credits page is posted, a new batch of form entries roll in and several students issue corrections. Now, not only does Alice have to manually re-evaluate her workflow, she also needs to manually merge her edits (made in the spreadsheet) with the updated outputs — a tedious and error-prone process.

> **Takeaway:** Data science is continuous; workflows, whether containing manual steps or not, will inevitably need to be re-applied in new settings, e.g., when some of the input data is updated.

1.2 Desired Interaction Modalities

Users interacting with data employ a mix of tools at each stage of a data-centric workflow's life cycle: (i) *Discovery*, where users identify relevant datasets; (ii) *Exploration*, where users become familiar with the data's structure and semantics, and identify potential problems; (iii) *Curation*, where users address data problems, integrate data, and transform data into a desired form; (iv) *Analysis*, where users answer questions and/or build models; and (v) *Deployment*, where users refactor the workflow into a form suitable for automation and/or production use. We emphasize that progress through these stages is not monotone: users frequently identify errors or shortcomings, and go through a (vi) *Debugging* process that usually results in a return to an earlier stage of development.

Artifact Manipulation. The first four stages primarily concern themselves with data *artifacts* like datasets, machine learning models, and data visualizations. Depending on the specific task, users might interact with these artifacts through: (i) *Scripts* in languages like Python, R, Scala, or SQL which are general-purpose tools that excel at bulk transformations that follow specific patterns; (ii) *Direct Manipulation Interfaces* like spreadsheets that enable the user to directly inspect and manipulate the data, making them well suited for small datasets, quick one-off repairs, and data entry tasks; or (iii) *Automated Tools* or libraries like command-line utilities, data curation and cleaning algorithms, and data visualizers that each perform one specific task (e.g., the creation or transformation of an artifact) sufficiently well to cover most usage. All of these modalities act on one or more source artifacts, and produce one or more output artifacts as a result.

Workflow Inspection. All stages, but specifically the Debugging and Deployment stages, require a holistic view of the workflow: (i) *Workflow Management* provides users with a way to trace back through their exploration process whether through a full workflow management system like Vistrails [4], a notebook system like Jupyter, or a provenance capture system like CamFlow [33] or ReproZip [11]; (ii) *Versioning* through revision control systems like GIT, or through 'Track Changes' features allows users to trace back through time to earlier revisions of the data and workflows; (iii) *Automatic Data Documentation* tools like profilers, inference of semantic types, and uncertainty trackers help users to understand and develop specific artifacts in the context of the broader workflow [10, 15, 16, 26, 37]. (iv) *Debuggers* and IDEs like PyCharm, or separate tools like PigPen [31] give users a view of internal system state, helping them to trace forward through their code, and streamline the process of refactoring. A common theme to all of these modalities is that they provide a way for users to trace (whether forward or backward) through the execution of their workflow and its state.

1.3 Requirements

We now outline how the modalities exemplified in the prior section translate into concrete requirements for Vizier. To make our modularity goals concrete, we focus on extensibility in terms of support for new modalities for Artifact Manipulation, as well as Workflow Inspection.

Workflow Inspection. The first requirement is forced by the objective itself. At a minimum we need a way to record the steps the user took to achieve her goal.

Requirement W1: *Vizier must be able to capture the steps (workflow) the user follows to produce an output artifact.*

State, i.e., artifacts created / used by a workflow, plays an important part as well, and we need to track the evolution of state as it flows through the workflow (step versioning) and over the evolution of the workflow itself (workflow versioning).

Requirement W2: *Vizier must support versioned workflows and both step and workflow versioning of artifacts.*

For each version of a workflow, there exist corresponding versions of artifacts produced by Vizier's workflow semantics (that we will introduce in Section 1.4). Vizier needs to ensure that artifacts are automatically refreshed to reflect the latest version of the workflow to avoid bugs and non-reproducible workflows [34].

Requirement W3: *Vizier needs to identify (and update) artifacts invalidated by a revision to the workflow.*

Artifact Manipulation. To provide a viable alternative to using a mix of tools in a data-oriented workflow that each implement one particular modality, Vizier has to be a platform that can host a multitude of modalities.

Requirement A1: *Vizier must support multiple modalities for interacting with data artifacts*

The remaining requirements arise from the diviersity of modalities. Crucially, we wish to decouple modalities from the semantics of the artifacts they create or manipulate.

Requirement A2: *Vizier needs standard data formats and APIs for exchanging artifacts between modalities.*

We want to allow users and automated workflow steps to attach annotations to artifacts or their component parts as a way to document uncertainty or ambiguity arising from the data or its preparation. However, general purpose annotation management systems like Mondrian [18] or DBNotes [12] do not take any annotation-specific semantics into account. For example, we should not propagate an annotation marking a value A as uncertainty that is used in a computation $A \vee B$ if B is guaranteed to be true.

Requirement A3: *Vizier should support the efficient propagation of annotations on the component parts of an artifact through annotation-specific semantics including semantics for uncertainty management.*

Some modalities such as spreadsheets allow (manual) updates to data artifacts. If the original data changes, then it should be possible to automatically re-apply these edits to the modified data.

Requirement A4: *Manual updates to an artifact must be re-applicable when the artifact is updated upstream.*

1.4 Solution Overview

An overview of Vizier's architecture is shown in Figure 1. Addressing requirement **W1**, the central abstraction in Vizier is a workflow: a linear sequence of steps. Unlike classical workflow systems, Vizier does not require users to explicitly declare information flow between steps. Rather Vizier borrows the model employed in popular computational notebooks like Jupyter, where inter-cell communication occurs through a global state (artifacts) passed sequentially through steps. Following notebook conventions, we refer to these steps as *cells*, and the global state as a *scope*, a map from artifact name to the version of the artifact valid at this point in the workflow. Vizier stores artifacts in common formats through a versioned **Artifact Store** (Section 2), addressing requirement **A2**. In Section 3, we formalize Vizier's workflow model, and show how we satisfy requirement **W3** by instrumenting how each cell interacts with the scope, allowing us to determine what artifact versions are valid.

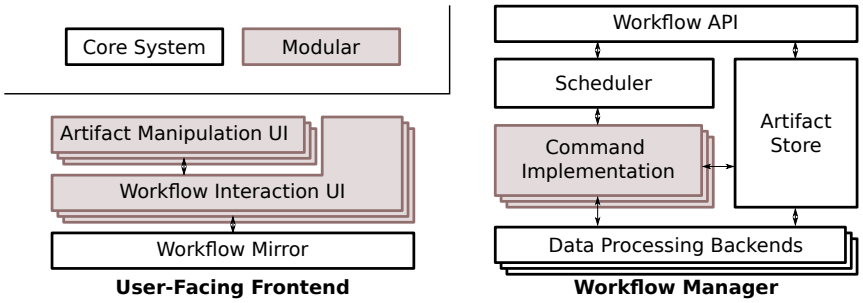


Figure 1: Vizier's architecture, comprised of a user-facing frontend component and a backend component.

Vizier's workflow semantics, paired with the versioned artifact store and workflow versioning (Section 3.2) addresses requirement **W2**. In contrast, classical notebooks like Jupyter or Zeppelin rely on the global state of an interpreter for inter-cell communication. Reverting this state to an earlier revision is challenging [39], limiting their ability to satisfy requirement **W3**. Vizier instead relies on its versioning system, allowing its **Scheduler** to automatically detect and re-evaluate stale cells (Section 3.3). To address requirement **A3**, we designed a light-weight uncertain data model that is implemented in Vizier in the form of *caveats*, annotations on data that indicate uncertain values and rows (Section 6).

Addressing requirement **A1** requires modularity in both Vizier's front- and back-end components. First, the user's interactions with a workflow and artifacts, whether through a scripting language, graphical interaction, or any other modality, need to be captured for replay (simultaneously addressing requirement **A4**). In Vizier this is achieved by requiring that every update to an artifact made through a particular modality has to be reflected as an operation in the workflow, i.e., a data update is translated into a workflow update. Vizier manages a collection of **Command Implementations** that implement the logic behind these artifact transformations (Section 4). To streamline the implementation of commands, Vizier's data formats and transformations are built over standard **Data Processing Backends** like Apache Spark.

The frontend is implemented over a **Workflow Mirror** that uses websockets to reflect a live view of the workflow the user is editing. Vizier automatically derives a default **Artifact Manipulation User Interface** for its notebook interface from each command's parameter schemas. This interface suffices for many templated commands, but the frontend can be further extended to provide a more customized experience, for example for Spreadsheet-style direct manipulation of data (Section 5). As illustrated in Figure 2, the frontend displays three **Workflow Interaction User Interfaces** by default: (i) A direct display of the workflow as a notebook, (ii) a table of contents summary of the notebook, including highlighting from documentation, and (iii) a list of artifacts derived by the notebook. Several of these components, including the notebook and the artifact list provide access to direct manipulation interfaces. Additional views currently implemented in Vizier include: (iv) A caveat view (Section 6) that shows and tracks potential errors in the workflow and data, (v) a history view that shows the evolution of the workflow over time, and (vi) a data provenance subway diagram view.

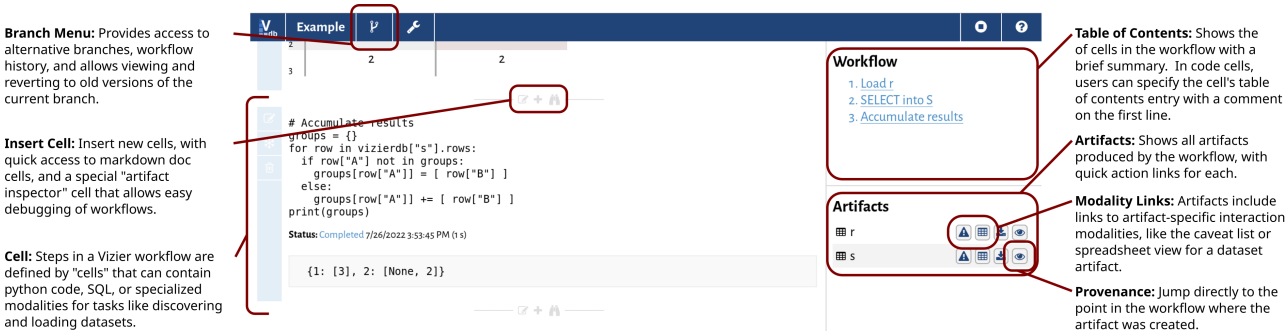


Figure 2: The Vizier User Interface

2 Versioned Data Artifact Store

Steps in Vizier workflows create, read, and update *"data artifacts"* or artifacts for short. To maximize interoperability between cells, Vizier establishes standards for how these artifacts are serialized, which we now discuss. Versions of artifacts are immutable. An update to an artifact creates a new object representing the updated artifact. Immutability greatly simplifies handling of state in Vizier and enables us to share artifact versions across multiple revisions of a workflow.

Dataset Artifacts. Tabular data is represented by Vizier as a Spark dataframe [3], a logical encoding of how a dataset is derived from source data. The choice to store datasets through a logical representation (specifying the computation instead of its result) is driven largely by the need for managing annotations (requirement **A3** which is implemented by rewriting the computation to handle annotations), but also results in lower space consumption.

Parameter Artifacts. A parameter artifact can be any primitive value of any data type supported by Apache Spark; We note that this includes simple nested collections like Arrays and Maps. This type of artifact provides a way to parameterize scripts and other system components, particularly for non-technical users. and parameter artifacts are used to pass simple data (e.g., simple constants in Python) between cells.

In addition to these two artifact types, Vizier also supports plots (stored in the vegalite [36] format), blobs and uninterpreted files, and language-specific constructs, e.g., Python function definitions. For lack of space, we do not further discuss these other artifact types.

3 Workflow Provenance Model and Runtime

In this section, we outline Vizier's workflow and versioning model, how we determine which cells of a workflow need to be re-executed if a cell is changed, and introduce the parallel scheduler of the system.

3.1 Workflow Model

A workflow in Vizier is a sequence of workflow steps (cells) that can read and write artifacts. Artifacts are versioned at the granularity of cells. A cell writing an artifact a causes a new version of a to be created. We will discuss the APIs Vizier exposes to the data interaction modalities for accessing artifacts in Section 4. The semantics of a Vizier workflow is the serial execution of the cells of the workflow, where the latest version of each artifact is passed as input to the cell that will be executed next. Thus, as long as the cells themselves are deterministic, the artifact versions created by the execution of a workflow are uniquely determined by the workflow itself.

```
data = read_csv("social_data.csv")
show(data)
```

```
data["latlon"] = geocode(data["address"])
show(data)
```

```
censusblocks = read_geojson("blocks.json")
show(censusblocks)
```

```
data = spatial_join(data, censusblocks, ["latlon", "geometry"])
count_per_block = data.groupby("block_id").count()
show(count_per_block)
```

Figure 3: A simplified example notebook

Example 3.1: Figure 3 shows a notebook with a simplified data ingestion, cleaning, and analysis task. The notebook loads the dataset (cell 1), geocodes listed street addresses (cell 2), loads a collection of census blocks (cell 3), and computes summary statistics for each census block (cell 4). We use this notebook to illustrate a key challenge of traditional notebook architectures. The user modifies cell 2 to switch to a different geocoder, potentially requiring re-evaluation of the notebook. In this specific notebook, the user had the foresight to write in an idempotent style, making it unnecessary to re-run cell 1 to recover the state needed to run cell 2 correctly. It is also unnecessary to re-run cell 3, as it does not depend on the output of cell 2. However, in traditional notebooks like Jupyter the burden of deciding which cells to re-evaluate rests on the user, creating added overhead and increasing the chance of errors.

Formally, a Vizier workflow is a sequence $\mathcal{N} = \{c_1, \ldots, c_N\}$ of cells. The versions of artifacts valid after execution of cell c_i are modeled as a global scope $\mathcal{G}$ that maps artifact names to artifact versions or the distinguished symbol $\perp$, which indicates that no version of the artifact has been produced yet. A cell c is a function that takes a scope $\mathcal{G}$ as input and produces an updated scope $\mathcal{G}'$: $c(\mathcal{G}) = \mathcal{G}'$. We use $\vec{r}_{\mathcal{G},i}$ to denote the names of artifacts accessed by cell c_i applied to $\mathcal{G}_i$. The scope parameter is necessary, because a cell may dynamically decide which artifacts to read based on the content of other artifacts. The result $[\![\mathcal{N}]\!]$ of evaluating workflow $\mathcal{N} = \{c_1, \ldots, c_N\}$ is defined as the scope $\mathcal{G}_n$ produced by starting with an empty scope $\mathcal{G}_0$ (where all artifact names are mapped to $\perp$), and by computing $\mathcal{G}_i = c_i(\mathcal{G}_{i-1})$.

If workflow $\mathcal{N}$ is modified by replacing a cell c_i with a cell c_i' (denoted as $\mathcal{N}[c_i \backslash c_i']$), we need to obtain the updated scope $[\![\mathcal{N}[c_i \backslash c_i']]\!]$. Of course this can be achieved by evaluating $\mathcal{N}[c_i \backslash c_i']$. However, to improve performance, Vizier attempts to update the output of $[\![\mathcal{N}]\!]$ by only re-evaluating a subset of the cells. First, observe that $\mathcal{G}_1, \ldots, \mathcal{G}_{i-1}$ are independent of c_i; and remain unchanged if c_i is modified. It is still necessary to have $\mathcal{G}_{i-1}$ to evaluate c_i'; so Vizier caches all intermediate global scopes after evaluating each workflow revision.

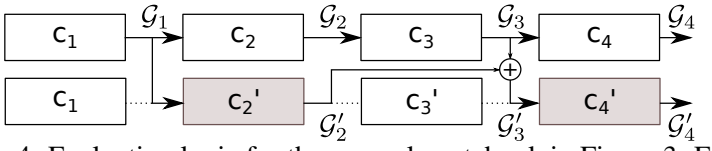


Figure 4: Evaluation logic for the example notebook in Figure 3. Edges are labeled with global scope versions. Cells that are (re-)evaluated are highlighted, dotted lines represent simulated state flow.

Naively, we have to still re-evaluate $c_{i+1}, \ldots, c_N$ using the new scope produced by c_i'. Let us denote the scopes produced by this evaluation as $\mathcal{G}'_{i+1}, \ldots, \mathcal{G}_n$. Vizier uses the readsets $\vec{r}_{\mathcal{G},i+1}, \ldots, \vec{r}_{\mathcal{G},N}$, to identify cells c_j for which the same output (artifact versions written by the cell) in $[\![\mathcal{N}[c_i \backslash c_i']]\!]$ is guaranteed. Denote by $\Delta(\mathcal{G}, \mathcal{G}') = \{k | \mathcal{G}(k) \neq \mathcal{G}'(k)\}$ the names of artifacts that differ between $\mathcal{G}$ and $\mathcal{G}'$. We have to re-execute cell

c_{i+1} if an artifact read by c_{i+1} has changed, which is the case if $\Delta(\mathcal{G}_i, \mathcal{G}'_i) \cap \vec{r}_{\mathcal{G},i+1} \neq \emptyset$. If c_{i+1} needs to be re-executed, then we set $\mathcal{G}'_{i+1} = c_{i+1}(\mathcal{G}'_i)$. Otherwise, we generate $\mathcal{G}_{i+1}$ by merging the changes made by c_{i+1} in $\mathcal{N}$ to $\mathcal{G}_i$ into $\mathcal{G}'_i$. That is, for all artifacts k we set $\mathcal{G}'_{i+1}(k) = \mathcal{G}_{i+1}(k)$ if $k \in \Delta(\mathcal{G}_i, \mathcal{G}_{i+1})$ and $\mathcal{G}'_{i+1}(k) = \mathcal{G}'_i(k)$ otherwise. The same approach is applied to decide whether to evaluate or skip the remaining cells in $\mathcal{N}[c_i \backslash c_i{}']$.

Example 3.2: Figure 4 continues the running example from **??** 3.1. The user has replaced the initial version of cell c_2 with an updated version $c_2{}'$. The global scope produced by preceding cells ($\mathcal{G}_1$) may be re-used unchanged to evaluate $c_2{}'$, producing scope $\mathcal{G}'_2$. $\Delta(\mathcal{G}_2, \mathcal{G}'_2)$ is the `data` artifact, but the readset of c_3 is empty, and so this cell does not need to be re-evaluated. Instead $\mathcal{G}'_3$ is derived by merging variables the cell previously changed (i.e., $\Delta(\mathcal{G}_2, \mathcal{G}_3)$) into the current global scope. Finally, Vizier identifies that an element of $\vec{r}_{\mathcal{G},4}$ (the `data` variable) has changed between $\mathcal{G}_3$ and $\mathcal{G}'_3$, necessitating a re-evaluation of cell 4.

3.2 Versioning Workflows

Vizier maintains a branching history of the evolution of the notebook. A cell is further subdivided into (i) **cell metadata** (c_i) that is unique to the workflow (e.g., timestamps and execution status), (ii) **results** (r_i) of executing the cell, and (iii) a **module** (m_i) describing the command executed in the cell. The latter two components (results and modules) may be shared across workflows. A module object describes the command to be evaluated in a cell. This includes its type (e.g., Python or Scala script; SQL query; or one of the graphical widgets), as well as any parameters to the command (e.g., the script itself, or the artifact name to materialize query results as). A results object stores references to versions of artifacts in the artifact store created by the execution of the cell. We do not materialize the full global scope after each cell, but rather only the changes made to the global scope by the cell (i.e., the cell's write set). Any global scope $\mathcal{G}_i$ can be reconstructed from the preceding write sets.

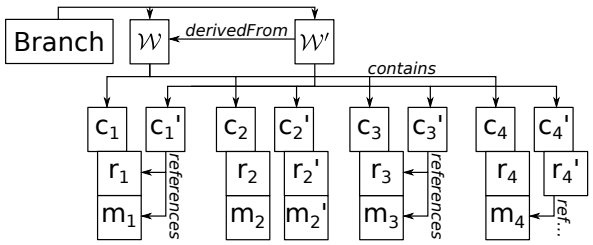


Figure 5: History for the running example, assuming cell 2 was changed as in Figure 4.

Example 3.3: Figure 5 continues our running example, which in our new terminology is a single branch comprised of two workflows $\mathcal{W}_1$ and $\mathcal{W}_2$. Both notebooks are comprised of four cell objects each. Cells 1, 3, and 4 were unchanged between workflows, and so the corresponding cell objects for $\mathcal{W}_2$ reference the same module description object as the corresponding cell for $\mathcal{W}_1$, while the module descriptions referenced by c_2 and $c_2{}'$ differ. Cells 1 and 3 do not need to be re-evaluated, and so can share their result objects, while Cells 2 and 4 both require re-evaluation and create new result objects.

Workflow and Branch History. Similar to version control systems like git, each workflow version in Vizier stores a reference to the workflow version it was derived from. A branch is an append-only sequence of workflow versions. The branch contains a reference to the most recent workflow version in the branch. A new branch may be created from any existing workflow version, including a historical one.

3.3 Parallel Scheduler

The evaluation model presented in Section 3.1 relies exclusively on *dynamic provenance*, where Vizier is informed about the readset and writeset of a cell at runtime when the cell accesses artifacts through Vizier's API. However, dynamic provenance is not always sufficient. For example, Vizier evaluates cells in parallel where possible, but dynamic provenance is not available until the cell has already been evaluated. Static provenance, which can derive a cell's read and write sets through static dataflow analysis of its source code, can be computed upfront.

However, static dataflow analysis is of necessity an approximation for some cell types; language features like control flow and dynamic code evaluation can lead to over- (or under-) estimates of the cell's read and write sets. Vizier uses a novel approach which refines static provenance at runtime using dynamic provenance [14].

Fundamentally, Vizier's scheduler needs to assign each cell in a running workflow to one of four lifecycle stages: (i) **DONE** when the cell has a valid result object, (ii) **PENDING** when the cell depends (directly or transitively) on a cell that does not have a result object, and (iii) **RUNNING** when the cell's dependencies have a valid result object but the cell itself does not. We further distinguish as (iv) **STALE** those cells in the **PENDING** stage for which we can conclusively determine that re-evaluation is required.

To manage lifecycle transitions, Vizier's scheduler relies on a combination of static and dynamic provenance [14]. It uses static provenance to generate an over-approximation on the read and write dependencies of a cell[2]. Accordingly, the scheduler tracks an over-approximation of the read and writesets at each step of the workflow, and refines them when the execution of cell finishes and we know its precise read and write set. This approximation is used by Vizier's scheduler to omit cells from re-execution or schedule them for parallel execution if we can determine that it is safe to do so based on these over-approximations.

4 Multimodality

As noted in Section 3.2, cells in Vizier implement a variety of different modalities, including scripting languages (e.g., Python, Scala), query languages (e.g., SQL), as well as graphical widgets for data ingestion, transformation, and curation (e.g., Load Dataset, Pivot Table, Repair Key). We refer to the evaluation logic for each modality as *cell command*. Each cell in a Vizier workflow identifies the command that should run to evaluate the cell, along with a set of arguments to that command. For example, the SQL cell (`sql.query`) takes two arguments: the text of a SQL query, and an optional name to materialize the result table as. In this section, we discuss how these modalities are implemented as cell types in Vizier.

A command is defined by the following methods: (i) **schema**: Returns a schema for the arguments the command accepts; (ii) **summary(arguments)**: Returns a textual description of the behavior of the command when parameterized by the provided arguments; (iii) **dependencies(arguments)**: Returns the over-approximation of the set of names of artifacts read (resp., written) by the cell as parameterized by the provided arguments, or indicates that either or both bounds can not be computed; and (iv) **evaluate(arguments, context)**: Evaluates the cell on the command, parameterized by the provided arguments and a context object.

The context object on which commands are evaluated provides a read/write interface to the global scope and a way to emit messages to be displayed alongside the cell in the notebook view. As we discuss in Section 3, the global scope maps artifact names to artifact versions. Artifacts are not persisted directly as part of the global scope, but rather are references to our write-only artifact store.[3] When a cell implementation writes an artifact through the context, the artifact version is serialized and written into the artifact store. The artifact store assigns the artifact (version) a unique identifier, which is then saved into the scope. Similarly, to read an artifact, a cell implementation first reads the artifact identifier out of the scope, and then accesses the corresponding artifact version from the store.

Cells implementing runtimes for general-purpose languages need to provide users of those languages with a way to interact with the global notebook state. This entails (i) providing a mechanism to reference the scope and import artifacts within a language-specific format, and (ii) predicting how the user-provided code will interact with the state to bound provenance.

[2]As we argue in [14], to deal with languages that support dynamic code evaluation such as Python, it would be necessary to allow under-approximations of read and write sets (missed data dependencies) and compensate for them at runtime. However, this not implemented in Vizier yet; attempts to dynamically read variables not in the maximal readset are flagged as errors.

[3]As mentioned before, artifact versions are immutable in Vizier which simplifies version management. We leave optimizations that selectively violate this policy and update artifacts or store deltas instead of creating full updated versions of artifacts to future work.

Python. Python cells are evaluated in an independent interpreter to avoid concurrency bottlenecks from the global interpreter lock (GIL). Thus, a key challenge is minimizing the volume of state transferred into and out of each interpreter. Global scope is lazily loaded by populating the global state with a set of proxy objects, one for each artifact in the scope. Access to the Vizier context for messaging and artifact access occurs through a control bus that, by default, operates over the python process' standard input and output streams. Vizier defines a special 'show' command within the module to produce structured messages (analogous to placing an item on the last line of a Jupyter cell). The show command includes support for most Vizier-defined types, matplotlib and bokeh plots, and provides fallbacks using the Jupyter-standard `_repr_html_` method, or direct stringification as a final resort. For predictive dependency tracking, and to identify variables that are modified, Vizier relies on Python's `ast` module, which provides an introspective compilation and code analysis framework. Vizier performs lightweight dependency analysis [14] to bound the cell's read and write dependencies.

SQL. SQL cells are parsed and evaluated by Apache Spark. Vizier intercepts the parsed SQL query AST and manually injects references to the corresponding artifacts — either datasets or python functions — where needed. Spark-provided view name decorators make it possible for the injected views to retain their names from the query, avoiding incomprehensible error messages. The same injection logic is also used to statically identify exact read and write dependencies.

5 Interactive Spreadsheets

A key feature of Vizier is support for direct interaction with artifacts [10], most notably a spreadsheet-like interface for interacting with Spark Data Frames. Spreadsheets provide a data exploration experience that is distinct from notebooks. Users are limited to a single dataset, but have significantly more flexibility when exploring the data. For example, it is common on small datasets (e.g., under 1000 rows) for users to complete preliminary data cleaning tasks like outlier detection, data integration, repair of typos and outliers, and even some limited computation (e.g., deriving new fields) in a spreadsheet prior to working with the data further (e.g., in a notebook). Another common use case is manual data entry; The user may enter the entire dataset, or may generate a data entry template (e.g., with a script) and import the resulting file into another tool.

Vizier's spreadsheet interface is intended to provide a view over a subset of the notebook, allowing users to interact with the dataset in the appropriate modality, while simultaneously preserving workflow provenance through interactions. As the user interacts with the spreadsheet, their edits are reflected in the notebook as new cells. As the user edits the notebook, their edits are likewise reflected in the spreadsheet — If the source data frame is updated in the notebook, Vizier attempts to re-apply the user's edits to the updated data.

Track Changes as a View. To support bi-directional interaction between spreadsheet and notebook views, Vizier implements a series of specialized cell types that mimic SQL DDL and DML operations, allowing for inserting, deleting, or reordering columns and rows, or for updating individual cell values. We refer to the operations described by these cell types collectively as the Vizual language [10, 17].

Vizual is based on the principle that the user's interactions with a database can be modeled as views over an original version of the data. As prior work has shown, a view defined over a table can mimic the effects of any DDL [13] or DML [30] operation applied to the table. Analogously, each Vizual cell in the notebook uses Spark's standard data frame manipulation language to apply a successive transformation to the dataset that mimics the user's interaction. To ensure that the spreadsheet remains responsive, a shim layer tentatively injects predicted updates to the user's interactions until the effects of the user's edit are fully applied (e.g., as [20]).

In SQL DML, update operations specify target rows by a predicate. By contrast, operations in a spreadsheet explicitly target specific rows of data, requiring Vizier to assign unique identifiers to each record to encode their order in the spreadsheet[4]. To allow Vizual operations to be replayed as source data changes, these identifiers

[4]We assume that the number of columns will remain manageable and reference them purely by name

should remain stable through data transformations. For derived data, Vizier uses a row identity model similar to GProM's [2] encoding of provenance. Derived rows, such as those produced by declaratively specified table updates, are identified by an appropriate combination of input tuple identifiers. For example, rows in the output of a join are identified by combining identifiers from the source rows that produced them into a single identifier, and rows in the output of a projection or selection use the identifier of the source row that produced them.

The remaining challenge is assigning row identifiers to source data, which we want to remain stable through changes to the source data so that spreadsheet operations can be replayed. Ideally, the data would include a unique identifier that we can leverage; but this is not always the case. Storing the data in a revision control system [9, 21], is not always a viable option [1]. A more heavyweight approach is to link records across revisions of a dataset [38], but this adds non-negligible overhead to common-case data revisions. Vizier presently supports persistent identifiers through append- or edit-only revisions by assigning each record a unique identifier based on its position, and a hash of its contents. This approach has the benefit of being lightweight (it can be applied in a single pass), and resilient. In contrast to simply using a hash-based identifier, the approach supports duplicate records. Conversely, solely using a position-based identifier could lead to spreadsheet operations being applied to the wrong row in case of insertions. While techniques for creating identifiers that are stable under updates has been studied extensively for XML databases (e.g., ORDPATH [32]) and recently also for spreadsheet views of relational databases [5], the main challenge we face in Vizier is how to retain row identity when a new version of a dataset is loaded into Vizier, as opposed to keeping identity consistent once the data is already in the system.

6 Data Documentation, Error, and Uncertainty Management with Caveats

Like notebook systems, Vizier enables users to document their workflow through markdown cells which do not manipulate artifacts, but simply serve as documentation. However, some documentation is specific to individual artifacts, or their component parts (e.g., rows, or columns); We would like such documentation to accompany the data as it is transformed [26]. Like other annotation management systems including Mondrian [18] and DBNotes [8], Vizier empowers users to annotate data with textual comments. However, in contrast to these systems, in Vizier these annotations also have a precise semantics: they encode uncertainty about an attribute value of a row or the existence of a row. This is important, because uncertainty arises naturally in most data science pipelines (e.g., because of errors in the data or because of heuristic choices during data cleaning) and if data analysis ignores the uncertainty in the data, it can lead to analysis results that cannot be trusted. To address this issue, caveats are propagated through operations on dataset artifacts in Vizier using an efficient uncertain query semantics we have developed [15, 16]. Thus, caveats on values and rows in the result of an analysis conducted using Vizier encode information about how data cleaning and curation operations on the data used in the analysis affect the analysis result. Furthermore, Vizier's implementation of data cleaning operations introduce caveats to encode information about other possible repairs.

6.1 Incomplete Databases

Formally, caveats in Vizier are based on an approximation of incomplete databases. An incomplete database $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of deterministic databases called possible worlds that encode alternative possibilities for the state of the real world: one possible world corresponds to the actual state of the real world, but we do not know which. As an example, consider an analyst that has to find the names of important customers (e.g., who ordered products totaling more than \$500). An example instance is shown in Figure 6. A common method for primary key repair is to group rows by their PK values and select one row from each group to be retained. However, typically we have insufficient information to know which row is the correct choice and will have to rely on heuristics (e.g., selecting the most recently updated row if this information is available). Incomplete databases can be used to model this uncertainty: we create an incomplete database whose worlds are all the repairs of the database violating the constraint [7]. Figure 6 shows two (out of 4) possible repairs for this dataset.

Customer Relation

cid	name	total
1	Peter Petersen	1000
1	Peter Petersen	950
2	Bob Smith	300
3	Alice Smith	400
3	Alice Smith	600

Certain Answers

name
Peter Petersen

Possible World (Repair) D_1

cid	name	total
1	Peter Petersen	1000
2	Bob Smith	300
3	Alice Smith	400

Answers in D_1

name
Peter Petersen

Possible World (Repqir) D_2

cid	name	total
1	Peter Petersen	1000
2	Bob Smith	300
3	Alice Smith	600

Answers in D_2

name
Peter Petersen
Alice Smith

Figure 6: Example customer database violating the primary key constraint that cid is unique and two possible worlds corresponding to some of the possible repairs of the database achieved by selecting one row among each group of rows with the same primary key value.

Typical constraint-repair algorithms will select one repair (one possible world) based on a heuristic like selecting the row whose values are most common in the dataset [35]. For instance, the cleaning algorithm may choose D_1 and the user would then evaluate their query (shown below) over D_1.

```
SELECT name FROM Customer WHERE total > 500;
```

While such heuristics may be quite effective on average and are certainly superior to just randomly selecting a world, it is unavoidable that they fail for some cleaning scenarios. An alternative approach called consistent query answering [6] takes a conservative stance, instead of selecting one repair, we reason about all possible repairs and only return query answers (the so-called *certain answers*) that are in the query's results for every repair (i.e., are guaranteed to be in the result independent of which repair is correct). This approach has the advantage that only correct query answers are returned, but is computationally expensive, may exclude many very likely answers (if they are not 100% certain), and is not closed (it is not possible to evaluate queries with certain answer semantics over the certain answers of a query).

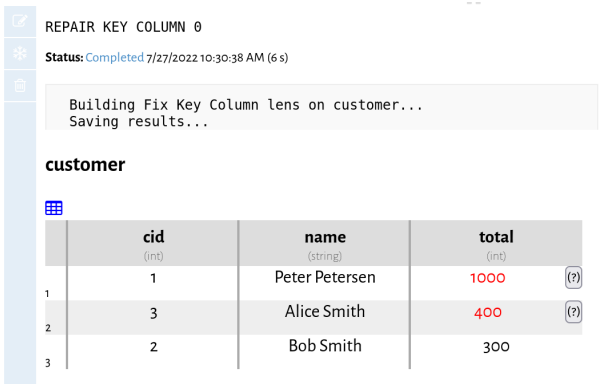


Figure 7: A UB-DB in Vizier encoding D_1 with possibly uncertain cells marked with caveats.

6.2 Attribute- and Row-level Caveats and Uncertainty-Annotated Databases

For Vizier, we developed an uncertain data model called uncertainty-annotated databases [15] that annotates one possible world (the so-called *selected-guess world*) with an under-approximation of certain answers (if we claim that a row is certain, then it is certain) that can be computed efficiently. This is encoded by annotating a subset of a dataset's rows with row-level caveats to mark them as not being certain. The reason that we use an under-

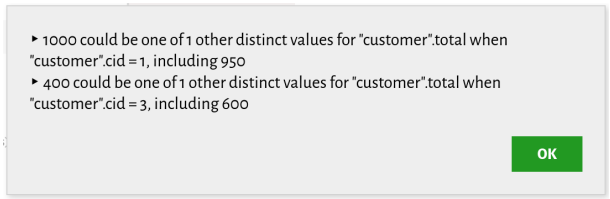


Figure 8: Caveats annotating the relation D_1

approximation is to be able to evaluate complex queries (full relational algebra including aggregation) efficiently (with PTIME data complexity and small overhead over deterministic query processing). Furthermore, attribute-level caveats are used to mark attribute values as uncertain (they may not be the same in every possible world) which is similar in nature to certain answers with nulls [27].

7 Conclusions and Future Work

In this paper, we have made the case for a multi-modal data science platform built on top of an incremental, data-centric workflow engine and have introduced the reader to Vizier, our system implementing this vision. Because each modality (notebooks, spreadsheets, the caveat view, etc...) is a "view" interacting with the same underlying workflow and datasets, it is easy to extend the system to support new interaction paradigms in the future. Building our system so that cells execute in isolation and only interact through dataflow makes it easy to add new cell types to the system, because we only have to worry about the interaction of the new cell type with data artifacts. Thus, adding support for other interaction modalities (e.g., new programming languages) into the system is straight-forward: we implement a new cell type and an API to access Vizier artifacts from within the modality.

As demonstrated in recent preliminary work, having a scheduler that revises its schedule based on dependencies discovered at runtime and using static dataflow analysis, we can execute notebook cells in parallel and avoid re-executing cells that are guaranteed not to change during automatic refresh. Significantly more opportunities for avoiding work exist, in particular by leveraging Vizier's standardized representation of artifacts. For example, by representing updates to dataset artifacts as change sets, existing approaches for incremental view maintenance can reduce the runtime overhead of recomputing workflow steps, as well as the space costs of preserving multiple artifact versions. Another interesting direction for future work is to study incremental maintenance of workflow results when the definition of a workflow step changes. This is an novel variation of the traditional incremental view maintenance problem where we have to update the result based on a change to the query rather than based on a change to the data.

Similarly, standard representations of artifacts can be used to help users better understand the outputs of their workflows. For example, numerous efforts have explored causal explanations in database query results ([19, 23, 28, 29], and explainability in machine learning has recently become an area of active research. For such techniques to be truly valuable, they need to operate across artifact types. For example, what would it take to link a sudden change in predictions made by a model to the addition of a new category in source data five transformations removed from the model training step. With Vizier's uncertainty model and provenance tracking we lay the ground work to develop methods for generating explanations that span multiple steps in a workflow.

Finally, we note that Vizier provides a "ground-up" approach to stitching different interaction modalities into a single workflow. Tools implemented as views over Vizier's workflow model seamlessly interact with each other, with coarse-grained provenance, reactive cell execution, and repeatability/reproducibility. However, these capabilities are limited to a single Vizier instance, and are of limited value to already existing tools. An important open challenge is the design of a federated infrastructure for data analytics workflows, allowing multiple Vizier instances or unrelated tools to interoperate.

Acknowledgements. This work was supported by NSF Awards ACI-1640864, IIS-1750460, IIS-1956149, and IIS-2125516.

References

[1] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. In *SIGMOD Conference*, pages 241–252. ACM, 2012.

[2] B. S. Arab, S. Feng, B. Glavic, S. Lee, X. Niu, and Q. Zeng. Gprom - A swiss army knife for your provenance needs. *IEEE Data Eng. Bull.*, 41(1):51–62, 2018.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the ACM International Conference on Management of Data (SIGMO)*, pages 1383–1394. ACM, 2015.

[4] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. Crossno, C. T. Silva, and J. Freire. Vistrails: Enabling interactive multiple-view visualizations. In *IEEE Visualization*, pages 135–142. IEEE Computer Society, 2005.

[5] M. Bendre, B. Sun, D. Zhang, X. Zhou, K. C. Chang, and A. G. Parameswaran. DATASPREAD: unifying databases and spreadsheets. *Proc. VLDB Endow.*, 8(12):2000–2003, 2015.

[6] L. Bertossi. Database repairing and consistent query answering. *Synthesis Lectures on Data Management*, 3(5):1–121, 2011.

[7] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin. Sampling from repairs of conditional functional dependency violations. *VLDB J.*, 23(1):103–128, 2014.

[8] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB J.*, 14(4):373–396, 2005.

[9] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *CIDR*. www.cidrdb.org, 2015.

[10] M. Brachmann, W. Spoth, O. Kennedy, B. Glavic, H. Müller, S. Castel, C. Bautista, and J. Freire. Your notebook is not crumby enough, replace it. In *Proceedings of the 10th Conference on Innovative Data Systems*, 2020.

[11] F. S. Chirigati, D. E. Shasha, and J. Freire. Reprozip: Using provenance to support computational reproducibility. In *TaPP*. USENIX Association, 2013.

[12] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *SIGMOD Conference*, pages 942–944. ACM, 2005.

[13] C. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the PRISM workbench. *Proc. VLDB Endow.*, 1(1):761–772, 2008.

[14] N. Deo, B. Glavic, and O. Kennedy. Runtime provenance refinement for notebooks. In *Proceedings of the 14th International Workshop on the Theory and Practice of Provenance*, 2022.

[15] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Uncertainty annotated databases - a lightweight approach for approximating certain answers. In *Proceedings of the 44th International Conference on Management of Data*, 2019.

[16] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Efficient uncertainty tracking for complex queries with attribute-level bounds. In *Proceedings of the 46th International Conference on Management of Data*, page 528 – 540, 2021.

[17] J. Freire, B. Glavic, O. Kennedy, and H. Müller. The Exception that Improves the Rule. In *SIGMOD Workshop on Human-In-the-Loop Data Analytics*, 2016.

[18] F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: Annotating and Querying Databases through Colors and Blocks. Technical report, University of Edinburgh, 2005.

[19] B. Glavic, A. Meliou, and S. Roy. Trends in explanations: Understanding and debugging data-driven systems. *Found. Trends Databases*, 11(3):226–318, 2021.

[20] N. Gupta, A. J. Demers, J. Gehrke, P. Unterbrunner, and W. M. White. Scalability for virtual worlds. In *ICDE*, pages 1311–1314. IEEE Computer Society, 2009.

[21] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. G. Parameswaran. Orpheusdb: Bolt-on versioning for relational databases. *Proc. VLDB Endow.*, 10(10):1130–1141, 2017.

[22] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Found. Trends Databases*, 5(4):281–393, 2015.

[23] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD Conference*, pages 841–852. ACM, 2011.

[24] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372. ACM, 2011.

[25] P. Kumari, S. Achmiz, and O. Kennedy. Communicating data quality in on-demand curation. In *QDB*, 2016.

[26] P. Kumari, M. Brachmann, O. Kennedy, S. Feng, and B. Glavic. Datasense: Display agnostic data documentation. In *CIDR*, 2021.

[27] L. Libkin. Sql's three-valued logic and certain answers. *ACM Transactions on Database Systems (TODS)*, 41(1):1, 2016.

[28] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endow.*, 4(1):34–45, 2010.

[29] A. Meliou, S. Roy, and D. Suciu. Causality and explanations in databases. *Proc. VLDB Endow.*, 7(13):1715–1716, 2014.

[30] X. Niu, B. S. Arab, S. Lee, S. Feng, X. Zou, D. Gawlick, V. Krishnaswamy, Z. H. Liu, and B. Glavic. Debugging transactions and tracking their provenance with reenactment. *Proc. VLDB Endow.*, 10(12):1857–1860, 2017.

[31] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110. ACM, 2008.

[32] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: Insert-friendly XML node labels. In *SIGMOD Conference*, pages 903–908. ACM, 2004.

[33] T. F. J. Pasquier, X. Han, M. Goldstein, T. Moyer, D. M. Eyers, M. I. Seltzer, and J. Bacon. Practical whole-system provenance capture. In *SoCC*, pages 405–418. ACM, 2017.

[34] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In M. D. Storey, B. Adams, and S. Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada.*, pages 507–517. IEEE / ACM, 2019.

[35] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[36] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Vis. Comput. Graph.*, 23(1):341–350, 2017.

[37] Y. Yang, N. Meneghetti, R. Fehling, Z. H. Liu, and O. Kennedy. Lenses: an on-demand approach to etl. *Proceedings of the VLDB Endowment*, 8(12):1578–1589, 2015.

[38] G. S. Yilmaz, T. Wattanawaroon, L. Xu, A. Nigam, A. J. Elmore, and A. G. Parameswaran. Datadiff: User-interpretable data transformation summaries for collaborative data analysis. In *SIGMOD Conference*, pages 1769–1772. ACM, 2018.

[39] K. Zielnicki. Nodebook. `https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/`.

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
Renmin University of China
Beijing 100872, China
duyong@ruc.edu.cn

TCDE Chair

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

IEEE Computer Society
10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1314

Non-profit Org.
U.S. Postage
PAID
Los Alamitos, CA
Permit 1398