DataPlay: Interactive Tweaking and Example-driven Correction of Graphical Database Queries

Azza Abouzied
Yale University
azza@cs.yale.edu

Joseph M. Hellerstein
Univ. of California, Berkeley
hellerstein@cs.berkeley.edu

Avi Silberschatz
Yale University
avi@cs.yale.edu

ABSTRACT

Writing complex queries in SQL is a challenge for users. Prior work has developed several techniques to ease query specification but none of these techniques are applicable to a particularly difficult class of queries: *quantified queries*. Our hypothesis is that users prefer to specify quantified queries interactively by *trial-and-error*. We identify two impediments to this form of interactive trial-and-error query specification in SQL: (i) changing quantifiers often requires global syntactical query restructuring, and (ii) the absence of *non-answers* from SQL's results makes verifying query correctness difficult. We remedy these issues with *DataPlay*, a query tool with an underlying graphical query language, a unique data model and a graphical interface. DataPlay provides two interaction features that support trial-and-error query specification. First, DataPlay allows users to *directly manipulate* a graphical query by changing quantifiers and modifying dependencies between constraints. Users receive real-time feedback in the form of updated answers and non-answers. Second, DataPlay can *auto-correct* a user's query, based on user feedback about which tuples to keep or drop from the answers and non-answers. We evaluated the effectiveness of each interaction feature with a user study and we found that direct query manipulation is more effective than auto-correction for simple queries but auto-correction is more effective than direct query manipulation for more complex queries.

Author Keywords

Quantification; Query Specification; Query Correction; Semantic fine-tuning

ACM Classification Keywords

H.5.2 Information interfaces and presentation: User Interfaces - Graphical user interfaces.

General Terms

Design, Human Factors, Languages

INTRODUCTION

Despite decades of work on language and interface design, many database users find it difficult to specify complex queries. Of all query tasks, users find non-trivial quantification to be most difficult [20, 19]. Contemporary query specification paradigms, such as query-by-example, visualization-driven querying and faceted search offer help with specifying simple query blocks, but they offer very little assistance for precisely those queries that are most difficult to specify — quantified queries. For those queries, users are stuck with the powerful but difficult database query language: SQL.

Quantified queries evaluate constraints over *sets of tuples* rather than individual tuples, to determine whether a set as a whole satisfies the query. Inherent in these queries are (i) the grouping of tuples into sets, and (ii) the binding of constraints with either existential or universal quantifiers. Existential quantifiers ensure that some tuple in the set satisfies the constraint, while universal quantifiers ensure that all tuples in the set satisfy the constraint.

People engage in casual quantified-query specification with one other on a regular basis. For example, when buying flowers, we might request a bouquet of 'some red and white roses'. Despite this request being somewhat underspecified, the florist will still try putting together a bouquet. He may add some red and white roses and then add a few lilies. If we wanted *only* roses, we clarify the misunderstanding and say 'only roses'. If we wanted only red or white roses, we might allow the lilies but object to pink roses. Such casual interactions suggest that we are accustomed to specifying quantified queries by trial-and-error, where we adjust our initial query specification by modifying quantifiers — only roses — or modifying dependencies between constraints — only red or white roses but no other rose colors[1].

What makes SQL — the de facto query language for quantified querying — challenging is that it discourages this trial-and-error approach to querying. Effective trial-and-error query specification depends on (i) the facility to incrementally refine an incorrect query through *small tweaks* and (ii) the ability to understand the effect of these tweaks. SQL, however, does not facilitate incremental query refinement because it lacks *syntax locality*: semantically small tweaks, such as changing a quantifier from existential to universal, usually result in large changes in the structure and syntax of a SQL query. Moreover, SQL does not present the complete effects of a query. While SQL presents *answers* or tuples that satisfy a query, it does not present *non-answers* or tuples that do not satisfy the query. Non-answers and answers together

UIST'12, October 7–10, 2012, Cambridge, Massachusetts, USA.

[1]The dependency here is between a constraint on flower type and a constraint on flower color.

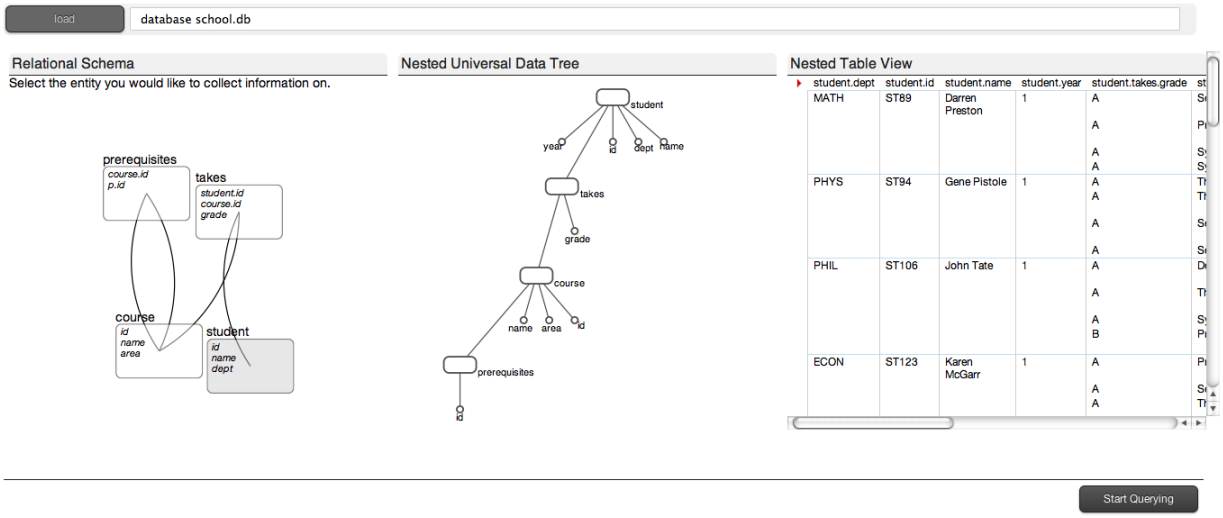

Figure 1. DataPlay's pivot interface: transforming a relational database into a nested data tree

help explain the behavior of quantifiers and help users assess the correctness of a query.

We present the design of *DataPlay*, a query tool that simplifies the specification and debugging of quantified queries by encouraging a trial-and-error approach to querying. DataPlay lets users specify their constraints without worrying about how to assemble them into a complete query. It builds from these constraints a graphical query, which users can *directly manipulate* to semantically refine the query. This is possible because the graphical query language exhibits syntax locality. DataPlay, also, helps users explore the space of query refinements by *visually suggesting possible manipulations*. As users tweak the query, it displays both *answers* and *non-answers* and keeps an *interactive graphical history viewer* of all modifications made to the query.

DataPlay also auto-corrects queries: users mark answers with 'want out' or 'keep in' labels and non-answers with 'want in' or 'keep out' labels and in turn DataPlay generates all modifications to the current query that satisfy the user's revision of answers and non-answers. To help users choose one query from several suggested queries, DataPlay rank-orders the generated queries by the size of their changes to the answers and non-answers, provides visual previews of these changes and also provides side-by-side comparisons of suggested queries.

DataPlay's *mixed-initiative user interface* integrates direct query manipulation with automated query correction. This interface is feasible because of DataPlay's unique data model and graphical query language. In this paper, we describe the data model and query language and how it relates to the relational data model and SQL.

Given the novelty of query auto-correction, we compare its effectiveness against direct query manipulation in a controlled user-study. We asked users to correct an incorrect query by either directly manipulating the graphical query or using the auto-correction feature. Our results show that users prefer direct manipulation to auto-correction for simple queries. As the query complexity increases, users prefer the query auto-correction feature; more users reach the correct query in less time. Thus the study provides evidence that a mixed-initiative query specification interface is superior to only a direct manipulation interface.

QUERYING WITH DATAPLAY

Querying with DataPlay involves four steps: (1) Pivoting, (2) Specifying constraints, (3) Fine-tuning and (4) Auto-correcting. Users typically iterate over steps two and three and occasionally auto-correct a query.

We explain how a user, Jane, performs each of these steps through an example query task. Suppose Jane has a school database and she wants to find students who
a) are in the CS department
b) completed *all* of CS11, CS16 and CS18 and
c) received A's in *all* three courses. She doesn't care about their grades in other courses.

Pivoting

Jane begins by connecting to the school database. DataPlay visualizes the database schema and the relationships between the tables of the database (left panel in Fig. 1). The school database consists of four tables: `student`, `course`, `takes` and `prerequisites`. The `student` and `course` tables represent tangible entities that users may wish to gather information about. Users will most-likely select one of these tables as a *pivot*. The remaining tables represent relationships between these entities. Since Jane is searching for students, she selects the `student` table as the pivot. DataPlay now restructures the database schema into a nested schema, which we call the *data tree* (middle panel Fig. 1). All tables are pivoted by the `student` table: for every student we have a nested-set of all the courses they took along with their grades in each course and for every course taken by a student we have a nested-set

Figure 2. DataPlay's query interface: (i) query tree (ii) interactive graphical history viewer (iii) command bar (iv) data and visualization panel

of all the prerequisites of that course. DataPlay thus transforms the many tables of the database into a single nested-universal table akin to a JSON or XML document. DataPlay shows Jane a tabular presentation of this nested-universal table to help her understand its nested structure (right panel in Fig. 1).

After pivoting, Jane begins querying the data. The query interface (Fig. 2) consists of (i) a query panel that visualizes the current query as a *query tree*, (ii) an interactive graphical history that presents all modifications made to a query, (iii) a command bar to specify constraints or visualizations in, and (iv) a data presentation panel which presents data-visualizations and two tables: an answer table and a non-answer table. At this point, all tuples of the nested-universal table are in the answers table and the non-answer table is empty.

Specifying Constraints

Jane can specify her constraints explicitly by writing simple propositions into a command bar or by brushing data-visualizations and having DataPlay automatically infer the propositions in the style of Tableau [3] or [8]. For the departmental constraint, she decides to visualize the department distribution first. As soon as she types 'dep' into a visualization-bar, DataPlay auto-suggests all attributes with 'dep'. She selects `student.dept`. DataPlay suggests two possible visualizations for the department attribute: a bar chart and a piechart. These suggestions are graphical thumbnails computed on a sample of the data. She picks the piechart; DataPlay adds a piechart to the visualization panel. She brushes the 'CS' pie; DataPlay (i) generates a constraint node with the propositional formula $\texttt{student.dept} =' \texttt{CS}'$, (ii) adds it on the query tree, (iii) executes the query and (iv) updates the answers and non-answers.

Jane then visualizes the `student.takes.grade` attribute in a bar chart. She brushes the A-bar. This causes DataPlay to add another constraint node with the proposition $\texttt{student.takes.grade} =' \texttt{A}'$. Since each student has a *set* of grades, DataPlay binds a default *existential quantifier* to the added constraint. The query tree finds CS students who have an A in their set of grades.

Jane types the following constraint for courses into the command bar:
$\texttt{student.takes.course.id}[= \texttt{CS11}, = \texttt{CS16}, = \texttt{CS18}]$
As a syntactic convenience, DataPlay enables users to specify multiple propositions in the above shorthand. DataPlay treats such propositions as a single constraint. As before, DataPlay generates another constraint node and adds it to the query tree and since each student takes multiple courses, DataPlay binds an existential quantifier to the course constraint.

Every modification made to the query results in the interactive update of answers and non-answers. In addition, a graphical snapshot of the preceding query tree is recorded in history. Jane can revert to a previous query tree by clicking its snapshot in the interactive history viewer; this adds the current tree to history before loading the previous query tree from history.

Jane has now specified all of her constraints and DataPlay has assembled the simplest possible query tree (Fig. 2), which finds students who are in the CS department, **and** took any of CS11, CS16 or CS18, **and** got at least one A in any course.

Fine-tuning

Jane starts fine-tuning the query tree. Any time she hovers over a constraint node on the query tree, DataPlay suggests possible manipulations to the node that will change the semantics of the query. In Figure 3, Jane hovers over the grade constraint and DataPlay suggests (i) toggling the quantifier from existential to universal such that students who *only* got A's are answers (Jane can click the 'Toggle Quantifier' button) or (ii) moving the grade constraint underneath the course constraint such that we only test for the presence of one of the three courses (CS11, CS16 and CS18) in the nested-set of courses taken that have an A grade. Jane drags the grade constraint node to the suggested position underneath the course constraint. She scans the answer and non-answer tables and is happy with the conditional dependence between the grade and course constraint: students who got A's in any of the three courses are answers. The query still needs fine-tuning; Jane wants students who got A's in all three courses. She decides to provide DataPlay with examples of tuples that are answers and tuples that are non-answers and let DataPlay fix her query for her.

Auto-correcting

In Figure 4, Jane marks the 'want out' checkbox for students in the answer table who didn't take all courses. She marks the 'keep in' checkbox for students who took all three courses with A's in them. She only marks a few tuples in the answer table. She decides not to mark any of the tuples in the non-answer table. Jane thus revises the result of the current query to match her intended query. She clicks the 'Fix my query' button. DataPlay constructs all query trees *derivable* from the current query tree and suggests query trees that satisfy the tuple memberships that Jane provided as possible corrections (See Section 'DataPlay's Query Auto-correction' for details).

DataPlay suggests only one correction for Jane's query: *covering* the course constraint. Coverage ensures that all propositions in a constraint are satisfied. Coverage is visually presented as a shading of the constraint node. Jane's accepts this correction and now the query finds all students in the CS department who took all three courses with A's in them.

FROM SQL DEFICIENCIES TO DATAPLAY FEATURES

We designed DataPlay out of a frustration with SQL query tools. We conducted an observational study on thirteen SQL users and analyzed what makes SQL challenging. We identified two key deficiencies, a lack of *syntax locality* and a lack of *non-answers*, which together hinder a trial-and-error approach to querying. We worked from SQL's deficiencies to DataPlay's features: our query language exhibits syntax locality and our data model enables the computation of non-answers for any query.

Figure 6 presents the results of the observational study. It shows the time it takes for trained[2] SQL users to specify a deceivingly complex query task: "find straight-A students

[2]On a 10-point scale 1 being basic knowledge and 10 being expert, the median self-reported SQL expertise was 5. The study participants were mainly graduate students in Systems research or undergraduate students who recently completed SQL training.

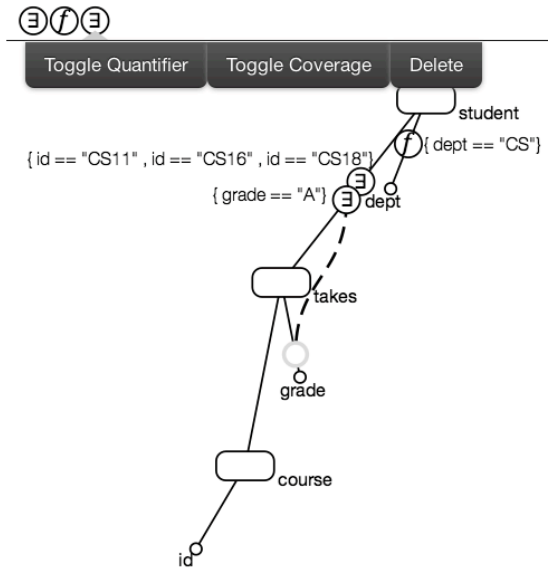

Figure 3. DataPlay visually suggests the different semantic modifications to a query's constraints

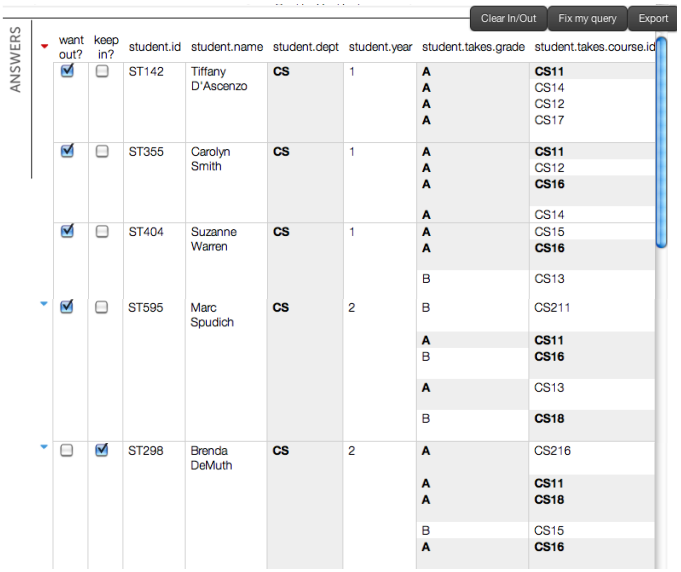

Figure 4. DataPlay's Correction by Example: users mark which tuples they want out or kept in Answers

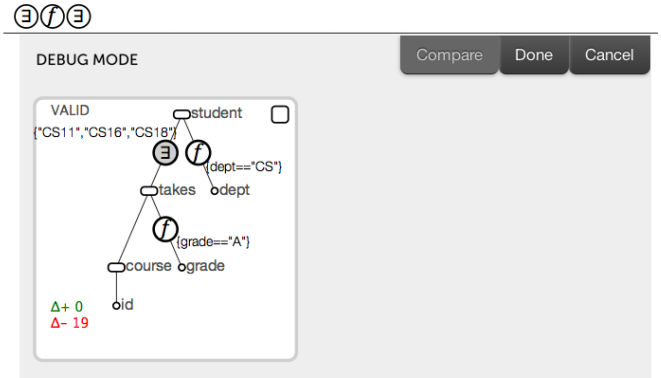

Figure 5. DataPlay searches the space of query trees derivable by manipulating the user's initial query to find query trees that satisfy the user's Answer/Non-Answer markings. Here, DataPlay found one correct query tree.

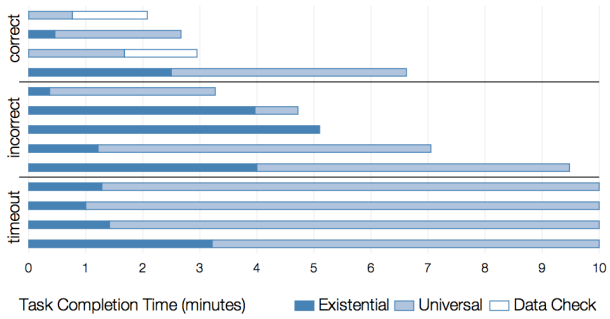

Figure 6. SQL query specification behavior of 13 SQL users.

(students who got A's in all their courses)". Only 4 of the 13 users successfully specified the query within 10 minutes! Most users (11 of 13) began with the simpler existential variant of this query, i.e., they found students with at least one A. This 'existential phase' was followed by a 'universal phase': users attempted to rewrite the existential query to get to the universal query. The prevalence of this two-phase querying behavior hints at the intrinsic trial-and-error nature of quantified querying.

In the following sections, we explain how each SQL deficiency discourages trial-and-error querying.

Syntax Locality

The following two SQL queries find students who got at least one A and students who got all A's respectively.

```
SELECT * FROM student s, takes t,
WHERE  t.grade = 'A' AND t.student_id = s.id;

SELECT * FROM student s, takes t
WHERE t.student_id = s.id AND s.id NOT IN
(SELECT student_id FROM takes WHERE grade != 'A');
```

A key observation about the SQL queries is that their structures are very different. The simple change of the quantifier from existential to universal appears to have global impact on the query syntax. For this reason we say that SQL often exhibits poor syntax locality with respect to changes in quantification. The inability to localize the details of the quantifier make the language confusing to users. Poor syntax locality discourages trial-and-error tuning of query specifications: small changes to query semantics can require large changes to query syntax.

Moreover, SQL users often feel *stuck* after formulating an incorrect query in SQL. Complex SQL queries tend to follow templates or tricks, which users learn with experience: two of the four users who wrote a correct SQL query were directly applying a learned template; they skipped the 'existential phase'. Users who haven't yet learned such templates cannot brute-force their way out of incorrect queries by applying different modifications.

Non-answers

As users write SQL queries, they look at the results to affirm that the written SQL matches their intended query. The presentation of a SQL query's results, however, is insufficient: two of the study's successful users wrote additional queries to check that their query was correct (Data Check phase in Figure 6): they arbitrarily chose a couple of students from the query result and wrote another query to view all the grades of these students to ensure that they indeed had no other non-A grades.

Existential

Answers

Nina Simone	BLUS101	A
Nina Simone	JAZZ101	A
Nina Simone	SOUL101	A
Bill Withers	BLUS101	A
Bill Withers	RYTH101	A

(a)

Non-Answers

Bill Withers	**CLAS101**	**C**
Louis Armstrong	REGA101	B
Bob Marley	BLUS101	C
Bob Marley	RYTM101	C
Bob Marley	JAZZ101	C

(b)

Universal

Answers

Nina Simone	BLUS101	A
Nina Simone	JAZZ101	A
Nina Simone	SOUL101	A
Frank Sinatra	CLAS101	A
Frank Sinatra	MELD101	A

(c)

Non-Answers

Bill Withers	BLUS101	**A**
Bill Withers	RYTH101	**A**
Bill Withers	CLAS101	C
Louis Armstrong	JAZZ101	**A**
Louis Armstrong	REGA101	B

(d)

Table 1. Answers and Non-Answers for queries that find students with at least one A and students with all A's

Tables 1a and 1c are example results of executing the existential and the universal SQL queries for finding students who got at least one A and students who got all As respectively.

Without any additional information, it is impossible to determine which query produced which of the sample results in Tables 1a and 1c. SQL interfaces usually provide us with the *answers* or the tuples that satisfy our query, but they do not provide us with the answers' complement: the *non-answers* or tuples that do not satisfy our query. Answers alone, however, may not help us fully understand a query. It is only when we see 'Bill Withers' in both the answers (Table 1a) and the non-answers (Table 1b), that we can deduce that Table 1a is the result of the *at least one A* query. Similarly, it is only when we see A's in the non-answers (Table 1d), that we can deduce that Table 1c is the result of the *straight-A* query.

Without non-answers, a user looking for straight-A students can mistakenly believe that an existentially-quantified query is correct just by examining the answers. Therefore, SQL interfaces that do not present non-answers hinder query interpretation and ultimately correct query specification.

Unfortunately, this problem is not amenable to a quick interface fix: complements are not commonplace in SQL interfaces because the pure relational query and data model make it hard to define and compute complements. Consider the following poorly constructed yet valid query:

```
SELECT * FROM student, takes WHERE
student.id = x AND takes.student_id = x;
```

In this query it is difficult to determine the *universe* from which we complement answers to form non-answers. It is likely that the user's perspective of the universe is `student` $\bowtie$ `takes`, which represents the courses taken by the student. However, the query has no apparent join and one can infer the

universe to be `student` $\times$ `takes`, which represents all possible combinations of student tuples with takes tuples. If we infer the second universe, we will compute a much larger and an incomprehensible non-answer set.

DataPlay's data model enables the presentation of non-answers and its graphical query language exhibits syntax locality. We describe the data model and query language in the following sections. We describe how the language's syntax locality and the presentation of answers and non-answers enable query auto-correction. We then discuss the inner workings of the query auto-correction feature.

DATAPLAY'S DATA MODEL

DataPlay restricts the relational data model to a *nested universal relation*: while this limits the expressive power of DataPlay compared to SQL to a degree, it enables a far more effective specification interface for a large class of sophisticated queries. We describe the data model, the features it enables and its limitations.

DataPlay uses a user-specified *pivot* to assemble a single nested universal table from the many tables of a database. We refer to the schema of this table as the *data tree*. Any relational schema can be transformed into a data tree: we map the schema of the relational database into a graph where nodes represent relations and edges represent primary-foreign key relationships between relations. We call this graph the *key-graph*. The *data tree* is a rooted spanning-tree of the key-graph with the pivot as the root. Starting with the pivot, we add it as the root of our data tree. We add all its attributes, excluding any foreign keys, as leaves. We then traverse all edges outgoing from the pivot in the key-graph in a depth-first fashion. For each relation that we add as a node to the data tree, we add all its attributes as leaves excluding foreign keys. In the data tree, every attribute is identified by its path from the pivot. We populate the nested universal table by taking the join of a child relation with a parent relation in the data tree.

The nested universal table is an abstract view on top of the relational database; we do need not to physically restructure the database or materialize multiple nested universal tables, one for every pivot.

While the nested universal table is not a new concept [13], it is a largely forgotten one. As the name suggests, it merges universal relations [14] with nested data models, such as JSON or XML. This buys us the following:

1. A natural grouping hierarchy for a large class of quantified-queries: For these queries, users are relieved from two cumbersome SQL steps: *join specification* and *group construction* through sub-queries.

2. A closed-world: Querying in DataPlay is simply the partitioning of the tuples of nested universal table into answers or non-answers. If tuples of the pivot satisfy all our constraints then they are answers, otherwise, they are non-answers. In contrast, non-answers, in the traditional relational-model, are neither well defined nor easy to compute as we discussed in the previous section.

DATAPLAY'S GRAPHICAL QUERY LANGUAGE

A query in our language is called a *query tree*. The class of queries captured by query trees are *Boolean conjunctive quantified* queries over nested relations. Each quantified expression consists of conjunctions or Horn[3] statements over Boolean functions (these are the constraints) and only conjunctions of quantified expressions are allowed.

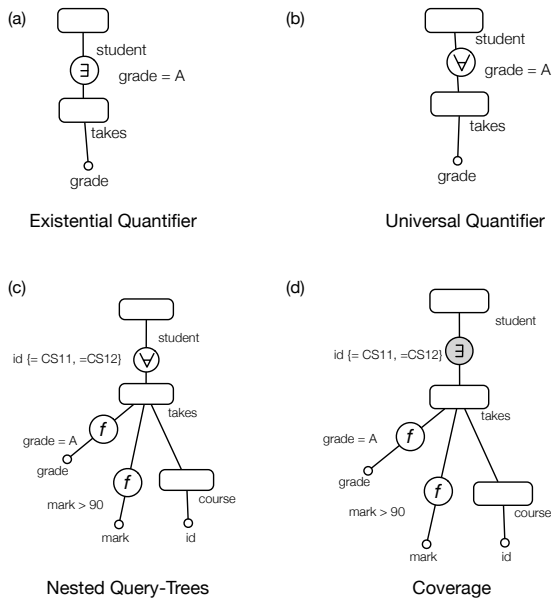

Figure 7. DataPlay's Graphical Query Language

The example query trees in Figure 7 comprehensively describe all aspects of DataPlay's graphical query language. Query trees are *data trees* overlaid with constraints. Every relation node is a query tree that maps its tuples to either answers or non-answers. An empty query tree — one without constraints — maps all its tuples to answers. The trees are conjunctive in nature, they take the conjunction of all constraints on their descendants —attributes or *answer* tuples from a nested query tree.

In Figure 7a, the empty query tree at `takes` maps all its tuples to answers as it has no constraints. For every student, the query tree at the pivot, `student`, determines whether there exists a tuple in the nested set of *answer*-tuples in `takes` where grade is A. Those students are answers. The remaining students are non-answers. In, Figure 7b, the query tree at the pivot, `student`, determines whether all grades of a student in the nested set of answer-tuples in `takes` are A.

Note that when a constraint node operates over a set of tuples, it has one of the following quantifier symbols in it: $\forall$ for the universal quantifier or $\exists$ for the existential quantifier. If the constraint node operates over an individual tuple and not a set of tuples, it requires no quantifier, and instead has the symbol f in it.

In Figure 7c, the query tree at `takes` maps tuples where grade is A and mark is above 90 to answers. The query tree at the

[3] e.g. $a \wedge b \rightarrow c$

`student` pivot, now tests if all of the nested answer tuples in `takes` are either CS11 or CS12 and no other courses. The universally-quantified course constraint here only examines the nested answer tuples of `takes` ignoring the nested non-answer tuples.

The query tree of Figure 7d illustrates a special feature of the language: *coverage*. Coverage allows users to ensure that *all* of a constraint's propositions (assuming it has more than one) are true for a set of tuples[4]. The query tree of Figure 7d ensures that a student has taken both CS11 and CS12 with A's and marks above 90. Coverage is denoted by a shading of the constraint node. Technically, coverage is a redundant feature of the language as one can write separate constraints for every course and the conjunctive nature of DataPlay will ensure coverage. However, users may conceptualize several propositions as a single self-contained constraint. Also, when brushing dense-visualizations, if we generate one constraint for every brushed area or glyph, rather than a list of propositions for a single constraint, we might over-crowd the query tree.

The graphical query language is expressive enough to describe a large class of quantified queries succinctly and with syntax locality: compare the query trees of Figure 7a and 7b with their equivalent SQL queries (see section on SQL 'Syntax Locality'). Notice that to change the semantics of a constraint from existential to universal, only a single symbol change is required. Users can tweak the properties of a constraint independently of other constraints and without re-creating a new query tree. Since each constraint supports a fixed set of manipulations (toggle quantifier, toggle coverage or change position), DataPlay visually suggests these manipulations to users. Users, no longer have to feel *stuck*: they can try out different manipulations until they reach their intended query.

Moreover, the language ensures that for a given collection of constraints, there is a finite and searchable space of possible query trees: this enables auto-correction. Instead of directly manipulating a query tree to achieve a certain partition of tuples into answers or non-answers, users can simply specify this partition by marking tuples as either answers or non-answers and DataPlay will search the space of query trees to find one that achieves the required partition.

Finally, compared to a textual query representation, query trees better reflect the hierarchical structure of the nested universal table and better illustrate the dependencies between constraints; nesting is visually awkward in a linear representation.

DATAPLAY'S QUERY AUTO-CORRECTION

We built the auto-correction feature because as the complexity of a query increases, the more difficult it becomes to debug and fix a query by directly manipulating it as we empirically demonstrate in the following section.

[4]A list of propositions is different from specifying a boolean conjunction or disjunction of these propositions: a list operates at the set level, while the disjunction or conjunction operates at an individual tuple-value.

Users validate a query's correctness by examining its answers and non-answers and searching for misplaced tuples. Enabling users to label offending tuples in answers with 'want out' and actual answers misplaced in non-answers with 'want in' and in turn correcting the query for the user reduces the *gulf of execution* [17] of a query specification tool as users combine validation and debugging in a single process. In DataPlay, we also allow user to label correctly placed answer and non-answer tuples with 'keep in' and 'keep out' respectively. We refer to the user-specified mapping of tuples into answers or non-answers as *tuple memberships*. Users only provide partial mappings of all database tuples to answers or non-answers: unmapped tuples are *free*. Automated query-correction must support such partial mappings as no user will map all tuples of a database to answers or non-answers.

DataPlay generates the space of all possible query trees derivable from the current query tree by (i) toggling quantifiers, (ii) toggling coverage or (iii) changing the positions of constraints. DataPlay suggests all query trees in this space that satisfy the given tuple memberships. If users provide a few or not enough *selective* tuple memberships the space of suggested query trees can be large and it might be difficult for users to understand the semantic differences between the suggested query trees. To reduce this *gulf of evaluation* [17]:

1. We preserve tuple memberships across several iterations of automated query correction. This allows users to pick any one of the suggested query trees and provide more tuple memberships if the selected query tree turns out to be incorrect. Users can explicitly clear tuple memberships by clicking a 'Clear In/Out' button.

2. We provide visual-previews of the effect of a suggested query tree: when users mouse-over over a suggested query tree, DataPlay shades tuples that are non-answers for that query tree.

3. We allow users to *diff* suggested query trees; DataPlay opens up a comparison window containing tuples on which the query trees disagree on whether they are answers or non-answers.

DataPlay rank-orders its query correction suggestions by the number of tuples that change membership from answers to non-answers or vice-versa. We chose this order under the assumption that users might directly manipulate a query to get close to their intended answers/non-answers and then use auto-correction to fix the membership of few problematic tuples. Thus, we order query trees that drastically change answers and non-answers last. This order, however, does not guarantee that the user's intended query is at the top of all the query tree suggestions.

EVALUATING DATAPLAY'S INTERFACE

As an initial evaluation of our mixed-initiative user interface, we conducted a comparative user study of DataPlay's novel query specification features: direct query manipulation and automated query correction. Our goal was to observe task completion times for each feature to determine which feature users were more effective at debugging queries with.

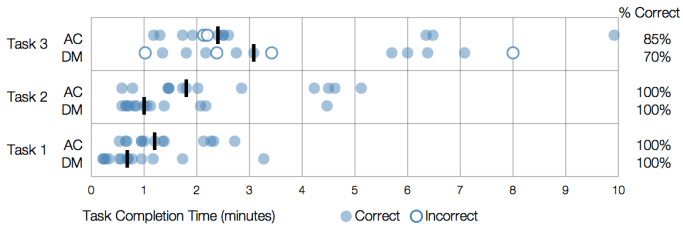

Figure 8. Black bars indicate median task completion times. 'DM' stands for Direct Manipulation and 'AC' for Auto-Correction.

Participants and Methods

We recruited the same 13 participants of the SQL observation study. These subjects are familiar with database querying: undergraduate students from an Introduction to Databases course and graduate students who regularly work with databases. The subjects had never used DataPlay before.

We first presented a 15-minute DataPlay video-tutorial. The tutorial described how to (i) load and pivot a database, (ii) specify constraints, (iii) directly manipulate a query tree and (iv) auto-correct a query tree by providing tuple memberships. We then allowed the participants to play with the tool for 10 minutes. The users directed this play-time and we answered any questions they had about the tool.

We then asked the subjects to fix a set of three incorrect queries of increasing complexity (described below) using only one of the features tested: direct query manipulation or automated query correction. Users then fixed a second set of three incorrect queries of similar complexity to the first three but on a different dataset and using the other, still unused feature. The choice of dataset for each feature was randomized across subjects. The two datasets were (randomly generated) student academic records and US flight arrival and departure records.

Task 1: We gave users a query tree that finds (a) students who got A's in some courses or (b) routes where some flights arrived late. We asked users to fix the query to find (a) students with all A's or (b) routes where all flights arrived late. The complexity of this task is '1-tweak'.

Task 2: We gave users a query tree that finds (a) students who took courses in any of three areas or (b) flights that were delayed by any of three causes. We asked users to fix the query to find (a) students who took courses in all and only the three areas or (b) flights that were delayed by all and only the three causes. The complexity of this task is '2-tweaks'.

Task 3: We gave users a query tree that finds (a) students who took any of three specific courses and got an A in any course or (b) flights that were delayed by any of three causes and some delays lasted more than 10 minutes. We asked users to fix the query to find (a) students who took all three courses with A's in them ignoring grades in other courses or (b) flights that were delayed by all three causes and for each cause the delay lasted more than 10 minutes. The complexity of this task is '3-tweaks'.

Both features are important

We performed a repeated-measures ANOVA of completion times with query complexity and feature used as independent factors. We log-transformed responses to better approximate a normal distribution. We found a significant main effect of query complexity ($F_{2,11} = 36.74, p < 0.001$). In Figure 8, we notice that overall speed and accuracy for both features are adversely affected as query complexity increases. We also found a significant main effect of feature ($F_{1,12} = 7.37, p < 0.02$) and a significant interaction effect of query complexity and feature ($F_{2,11} = 5.39, p = 0.02$); as the query complexity increased (Task 3), performance with the auto-correction tool improved over direct manipulation despite direct manipulation being significantly better than auto-correction for simpler query tasks.

In Figure 8, we observe that the median performance to fix the first two less complex query tasks with direct manipulation is faster than query correction. Users expressed through comments in a post-study questionnaire that when they knew exactly what changes they had to apply to the query tree, they felt query correction slowed them down because they had to spend more time scanning and labeling tuples.

For the third query task, however, we observe the opposite effect: the median performance with query correction is faster than direct manipulation. Even though only three tweaks are required to fix the third query, there were eleven possible query trees. With direct manipulation, users who were still unfamiliar with the query language had many query trees to manually search through, this slowed their overall time. One user expressed: "I really just bruce forced looking for the 2*2*2 options by toggling coverage, quantifier, and position". Those users performed better when they offloaded the mechanical brute-force search to DataPlay. Thus, with auto-correction, task completion time for 75% of the users was under 3 minutes, but with direct-manipulation only 46% of the users completed the task in under 3 minutes.

We also conducted a post-study questionnaire and asked users to rate the utility of DataPlay's features on a Likert scale: users found direct manipulation ($\mu = 4.5$) and query auto-correction ($\mu = 4.6$) to be extremely useful. Users were given the opportunity to elaborate on their ratings. One user explained that with auto-correction: "you no longer really even need to think in sql, just what you want or not [sic] want." Another explained that "auto-correction is useful, but I found marking tuples as 'want in'/'keep out' to be more tedious than [manually] correcting queries". Another user said "It was very useful in the sense that it was simple to manually filter the data elements. I would probably find myself using the manual constraint manipulation first to narrow down results and then using auto-correction if I became confused." As such, we believe that a mixed query specification interface is superior to an interface that only offers direct manipulation.

Users also found answers and non-answers to be very useful for debugging queries ($\mu = 4.6$), however they found the presentation of nested rows to be lacking: DataPlay by default collapses nested rows and shows the top four nested rows; users had to expand some rows to examine relevant

data. Users found DataPlay's highlighting of values that satisfy a constraint to be a useful feature when scanning answers and non-answers for correctness.

We note that the first query task in this study is as complex as the query task used in the SQL observational study. A qualitative argument in support of DataPlay can be made here: while only 2 of 11 users successfully transformed an existential SQL query to a universal one, all 13 users successfully transformed an existential query to a universal one in less than 3.5 minutes.

Reducing errors: DATAPLAY v2.0

As query complexity increased, the likelihood of errors also increased: 15% and 30% of users submitted incorrect queries for the third query task when using query auto-correction and direct manipulation respectively (See Fig. 8). We examined screen-recordings of users who failed the third query task and found the following two issues:

1. When users quickly scanned answers or non-answers, they sometimes missed tuples that indicate query-specification errors, especially if those tuples are not within the first few tuples.

2. When correcting queries, users may provide a few or not enough selective tuple-memberships: this results in DataPlay generating a large set of query trees that satisfy the given tuple-memberships. DataPlay's simple ordering of suggested query trees may not bring forward the user's intended query. Users, who submitted incorrect queries, provided a few non-selective tuple-memberships, and then picked the highest ranked query tree from DataPlay's suggestions and hastily concluded that they fixed the query.

The first issue can be resolved if we order answers and non-answers such that they *surface query errors* or at least cluster tuples that have identical data with respect to the constraints and present one tuple from each cluster, hence increasing the diversity of the top-k presented tuples. Alternatively, we can allow users to resample the top-k results, thus extracting more tuples from the database for further query verification.

The second issue is more complex and poses an exciting research challenge. Query trees map an extremely large *data space* to answers and non-answers, user-specified tuple memberships only map a small random sampling of the data space to answers and non-answers. To illustrate, with two constraints over two attributes, for example, grade = A and course.area = Systems, we have 2^2 or 4 possible satisfiability combinations for a two-attribute tuple:

Grade = A	Area = Systems	Example tuples
True	True	[A, Systems]
True	False	[A, Design]
False	True	[B, Systems]
False	False	[C, Theory]

If the attributes are two nesting levels deep then our data space is the number of subsets containing elements from the four possible two-attribute tuples, i.e 2^4 or 16. For example:

Student	Grade, Course Area	Pattern
John	A, Systems B, Design	True, True False, False
David	C, Design B, Systems	False, False False, True

With three constraints on attributes two nesting levels deep, the data space is 256 tuples and with four constraints, it is 65536 tuples. The double-exponential size of the data space means that many semantically-different query trees will satisfy the small sample of tuple memberships that users provide. Convergence to the one correct query may require an unreasonably large sample of tuple memberships: no user will provide tuple memberships for the entire space. Worse, the actual database may not be rich enough in its content and only contain a small sample of the data space. We are currently exploring techniques from learning theory to build an alternate interaction mode of query correction, where DataPlay constructs sample tuples in addition to using existing database tuples to interactively ask users tuple membership questions that guarantee convergence to one correct query tree within a *reasonable* number questions. Depending on the task, users provided as low as one tuple membership and as high as 19 tuple-memberships. The median number of tuple-memberships provided for each task was 4, 6 and 6. This data might help us define the bounds of what is reasonable.

In addition to reducing query specification errors, we would like to create more scalable query and data tree visualizations. In particular, we need to employ techniques such as interactive tree navigation and tree summarization to effectively visualize deeply-nested and bushy data trees (tables with many attributes).

RELATED WORK

DataPlay is influenced by and builds upon years of research on database data models and query languages. Our data model combines the universal relational model [14] with the nested relational model [13]. Our data model closely resembles that of the LORE database and our query language resembles LOREL, a non-graphical query language for nested data models [15].

DataPlay uses techniques from visualization-driven query tools [3, 18, 8]: users can brush visualizations in DataPlay to specify constraints. Similar to tools like Polaris [21] (and its commercial successor Tableau [3]) and Orion [9], DataPlay transforms user-interface actions into formal data-processing specifications. Polaris maps drag-and-drop operations on data variables into database queries and visualizations over single data tables [21]. Orion maps user-interface actions to data transformation statements and visualizations over network data [9].

The visual representation of queries in DataPlay as query trees is influenced in part by the success of code interfaces like Code Bubbles [4] and Stacksplorer [11] in enhancing code readability. These interfaces overlay a graphical 'call-graph' representation over code.

Recent research on *query causality* directed our attention

towards the presentation of non-answers. Query causality provides users with explanations for a particular query result [16]. Users pose questions such as "why is a tuple a non-answer?" and in return the database traces the behavior of the query to identify features of the tuple that make it a non-answer[5, 16]. It is from this research that we borrow the nomenclature of non-answers. We view query correction as the dual of query causality: while causality finds the data that caused a given result set, query correction finds the query that brings about a particular result.

Automated query correction is a novel feature of DataPlay. It applies techniques from *example-driven program synthesis*. While such techniques are currently used for data cleaning and integration [10, 7], we are not aware of any tool that applies such techniques for correcting a database query from example answers and non-answers.

We point out that Query-By-Example (QBE) [22] is not the same as automated query correction. QBE is a graphical *language* for relational data. It allows users to write queries by creating example table layouts and specifying constraints called condition boxes in the columns of these tables. Users find it difficult to specify quantified queries in QBE [19]; like SQL, QBE does not exhibit syntax locality and does not provide non-answers.

Commercial and open-source SQL graphical editors like Postgres' pgAdmin [2] and MS Access [1] enable users to build a graphical SQL query and directly manipulate it. With these editors, users physically draw the query by dropping tables into a drawing board and then connecting the tables with edges to represent joins. Direct manipulation is limited to removing or adding a table, selecting attributes to project and clicking edges to edit joins. Constraints are specified separately and not presented on the drawing board. In MS Access, constraints are specified using QBE. Users then execute the query to view answers only. These graphical editors do not address SQL's lack of syntax locality, thus users still need to learn SQL tricks and templates to specify quantified queries. DataPlay differs from these editors in many ways. DataPlay reverses the steps of query specification: users begin querying by directly specifying their constraints and DataPlay assembles the constraints into a query tree. DataPlay also interactively updates answers and non-answers and maintains a graphical history of all query modifications.

Like DataPlay, two recent research query tools are targeting the apparent gap in usable query tools for complex queries. *QueryViz*, translates SQL into an equivalent graphical language [6] to help users better understand a query, it does not enable direct query manipulation and it does not visually expose possible semantic modifications. SnipSuggest [12] helps users specify queries by auto-suggesting SQL clauses in a manner similar to code-assist tools in IDEs. SnipSuggest learns a variety of SQL templates from database users of a particular domain to provide context-aware suggestions ordered by popularity. SnipSuggest helps users of heavily-queried databases such as scientific data repositories by leveraging the work of the few initial users who write the complex queries. We can extend DataPlay such that, like SnipSuggest, it suggests popular constraints and it orders query trees by popularity during query correction.

CONCLUSION

In this paper, we introduced DataPlay, a query tool that simplifies the specification of quantified queries. We conducted a study to determine what makes quantified querying challenging in SQL. We observed that SQL hinders a trial and error approach to querying because it lacks syntax locality and it lack non-answers.

We designed DataPlay bearing in mind SQL's deficiencies. DataPlay's data model — a nested universal table — enables the computation of non-answers for any query and DataPlay's query language exhibits syntax locality. DataPlay's unique data model and query language help support several interaction features: (i) Users specify simple constraints without worrying about how to assemble these constraints into a whole query (ii) Users semantically tweak a query by directly manipulating a graphical query representation. The syntax locality of the language ensures that such tweaks are equivalent to small manipulations. (iii) Users asses the correctness of a query by examining answers and non-answers. Every query modification interactively updates answers and non-answers.

DataPlay provides automated query correction: users provide tuple memberships (examples of answer and non-answer tuples) and DataPlay suggests the possible manipulations to a given query that will satisfy the user's tuple memberships. We compared the effectiveness of direct manipulation with auto-correction through a user study. The study demonstrates that for simple queries, users perform better with direct manipulation, but as query complexity increases, users perform better with auto-correction.

REFERENCES

1. Microsoft access. `office.microsoft.com/en-us/access/`.
2. pgadmin: Postgresql administration and management tools. `www.pgadmin.org`.
3. Tableau software. `www.tableausoftware.com`.
4. Bragdon, A., et al. Code bubbles: a working set-based interface for code understanding and maintenance. In *CHI* (2010).
5. Chapman, A., and Jagadish, H. V. Why not? In *SIGMOD* (2009).
6. Danaparamita, J., and Gatterbauer, W. Queryviz: helping users understand sql queries and their patterns. In *EDBT/ICDT* (2011).
7. Gulwani, S., Harris, W. R., and Sing, R. Spreadsheet data manipulation using examples. In *CACM* (2012).
8. Heer, J., Agrawala, M., and Willett, W. Generalized selection via interactive query relaxation. CHI (2008).
9. Heer, J., and Perer, A. Orion: A system for modeling, transformation and visualization of multidimensional heterogeneous networks. In *VAST* (2011).

10. Kandel, S., Paepcke, A., Hellerstein, J., and Heer, J. Wrangler: Interactive visual specification of data transformation scripts. In *CHI* (2011).

11. Karrer, T., et al. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *UIST '11* (2011).

12. Khoussainova, N., Kwon, Y., Balazinska, M., and Suciu, D. Snipsuggest: context-aware autocompletion for sql. *Proc. VLDB Endow. 4*, 1 (2010).

13. Levene, M. *The nested universal relation database model*. Lecture notes in computer science. Springer-Verlag, 1990.

14. Maier, D., Ullman, J. D., and Vardi, M. Y. On the foundations of the universal relation model. *ACM Trans. Database Syst. 9*, 2 (June 1984).

15. McHugh, J., Abiteboul, S., Goldman, R., Quass, D., and Widom, J. Lore: a database management system for semistructured data. *SIGMOD Rec. 26*, 3 (Sept. 1997).

16. Meliou, A., Gatterbauer, W., Moore, K. F., and Suciu, D. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endow. 4*, 1 (Oct. 2010).

17. Norman, D. A. *The Design of Everyday Things*, reprint paperback ed. Basic Books, New York, 2002.

18. Olston, C., Stonebraker, M., Aiken, A., Aiken, E., and Hellerstein, J. M. Viqing: Visual interactive querying, 1998.

19. Reisner, P. Human factors studies of database query languages: A survey and assessment. *ACM Comput. Surv. 13*, 1 (1981), 13–31.

20. Reisner, P., Boyce, R. F., and Chamberlin, D. D. Human factors evaluation of two data base query languages: square and sequel. In *AFIPS '75*, ACM (New York, NY, USA, 1975), 447–452.

21. Stolte, C., and Hanrahan, P. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. In *INFOVIS* (2000).

22. Zloof, M. M. Query by example. In *AFIPS National Computer Conference* (1975), 431–438.