Playful Query Specification with DataPlay

Azza Abouzied
Yale University
azza@cs.yale.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

Avi Silberschatz
Yale University
avi@cs.yale.edu

ABSTRACT

DataPlay is a query tool that encourages a trial-and-error approach to query specification. DataPlay uses a graphical query language to make a particularly challenging query specification task - quantification - easier. It constrains the relational data model to enable the presentation of non-answers, in addition to answers, to aid query interpretation. Two novel features of DataPlay are suggesting semantic variations to a query and correcting queries by example. We introduce DataPlay as a sophisticated query specification tool and demonstrate its unique interaction models.

1. INTRODUCTION

Many database users find it difficult to specify complex queries, despite decades of work on language and interface design. Of all query tasks, users find non-trivial quantification to be most difficult [6, 5]. 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 query tasks that are most difficult—queries that go beyond the existential quantifiers implicit in simple select-from-where blocks.

The inherent intricacies of quantified queries necessitate a trial-and-error, incremental form of query specification. A query tool best supports this form of querying if it (i) enables semantic modifications to a query through small refinements and (ii) adequately presents the effects of these modifications. SQL, however, lacks *syntactic locality*; a simple quantifier change from existential to universal has a global impact on the query syntax. Thus, small semantic modifications require large changes to the query. Also, SQL interfaces lack an *observable complement*; while SQL interfaces present users with tuples that satisfy a query or the *answers*, they do not present users with tuples that do not satisfy the query or the *non-answers*. Non-answers, more than answers, help users understand the subtle effects of quantifier changes. Thus, SQL's presentation of the effects of semantic modifications can be inadequate.

We built DataPlay, a query tool that embodies the properties of syntactic locality and observable complements, to simplify query specification through the incremental fine-tuning of a query. The core of the tool is a graphical query language that visualizes all primary-foreign key relationships between attributes using a single *data tree*. The language reduces the task of query specification to that of correctly placing quantified constraints along edges of this tree. All features of a constraint are localized in its visual representation. DataPlay's query and data models are carefully chosen to guarantee that answers and non-answers for any query are well-defined and easy to compute. In addition, DataPlay supports the following novel features:

- Visual suggestions of semantic modifications to a query. Such suggestions encourage an interactive, trial-and-error form of query fine-tuning.
- A query correction tool: users mark answers with 'want out', '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.

We begin by illustrating how the lack of two key properties, syntactic locality and observable complements, makes SQL query specification hard (Section 2). We describe how DataPlay's data and query models achieve these properties (Section 3). We then describe how users specify queries with DataPlay (Section 4). We conclude with a description of our demonstration scenario (Section 5).

2. SQL'S FLAWS

2.1 Syntactic Locality

Consider the simple task of finding "A" students in a school database. This task is somewhat underspecified, and can be interpreted in a number of alternative ways; we will consider two. Listing 1a shows an existentially-quantified SQL query for finding students who got *at least one A*. Listing 1b shows a universally-quantified query for finding *straight-A* students—students who received no grades other than "A". One can imagine many other interpretations and variations.

The natural language specification of this query task highlights an important issue: quantification is often ambiguous, and as we translate from a casual specification to an explicit

Proceedings of the VLDB Endowment, Vol. 5, No. 12
Copyright 2012 VLDB Endowment 2150-8097/12/08... $ 10.00.

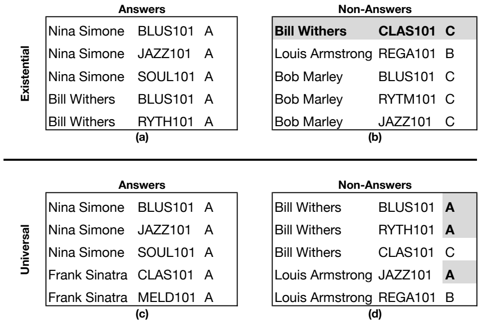
```
-- a) Some A.
SELECT * FROM student s, takes t,
WHERE   t.grade = 'A' AND t.student_id = s.id;

-- b) All A
SELECT * FROM student s JOIN takes t WHERE NOT
    EXISTS
(SELECT grade FROM takes WHERE grade != 'A'
AND student_id = s.id);
```

Listing 1: Quantified SQL queries

query syntax, we would like to "tweak" our query quantifiers to fine-tune the desired interpretation. To facilitate this form of query specification by incremental fine-tuning, tweaks need to be almost effortless. With SQL, however, a simple change of the quantifier from existential to universal appears to have a global impact on the query syntax as demonstrated in Listings 1a and 1b. For this reason we say that SQL often exhibits poor *syntactic locality* with respect to changes in quantification: small changes to query semantics can require large changes to query syntax.

2.2 Observable Complements

Let's extend our example with A-student queries even further. Consider the presentation of query results by a typical SQL interface. Tables 1a and 1c are example results for the *at least one A* query and the *straight-A* query.



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 each of the queries in Listing 1

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, rarely 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 lack an *observable complement* 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;
```

With this query, it is difficult to infer the user's perspective of the *universe* from which answers are extracted; the universe from the user's perspective is `student` ⋈ `takes` but since there are no joins in this query, one can infer the universe to be `student` × `takes`. An incorrect choice for the universe results in incomprehensible non-answers.

3. DATA AND QUERY MODEL

3.1 Data Model

DataPlay constraints its data model to enable well-defined and easy-to-understand complements. In particular, it uses the *nested universal relation* - nested UR [2]. As the name suggests, this model combines the properties of universal relations [3] with nested data models such as JSON, XML and nested relations [1, 4] to achieve two properties: (i) a single relation represents all the data in the database and (ii) all tuples in the relation are nested along a specific hierarchy.

Since, we have a closed world where a single relation represents the universe, answers are a subset of this universe and non-answers are the set-complement of answers.

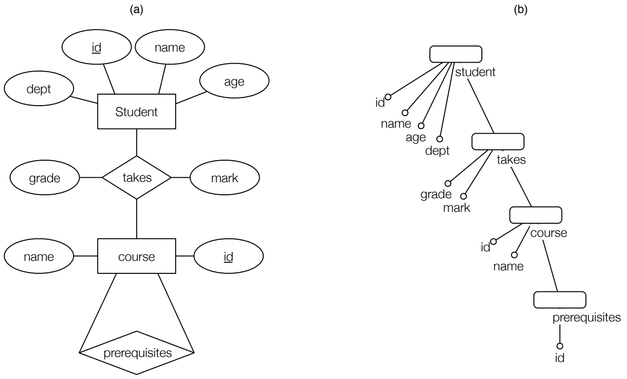
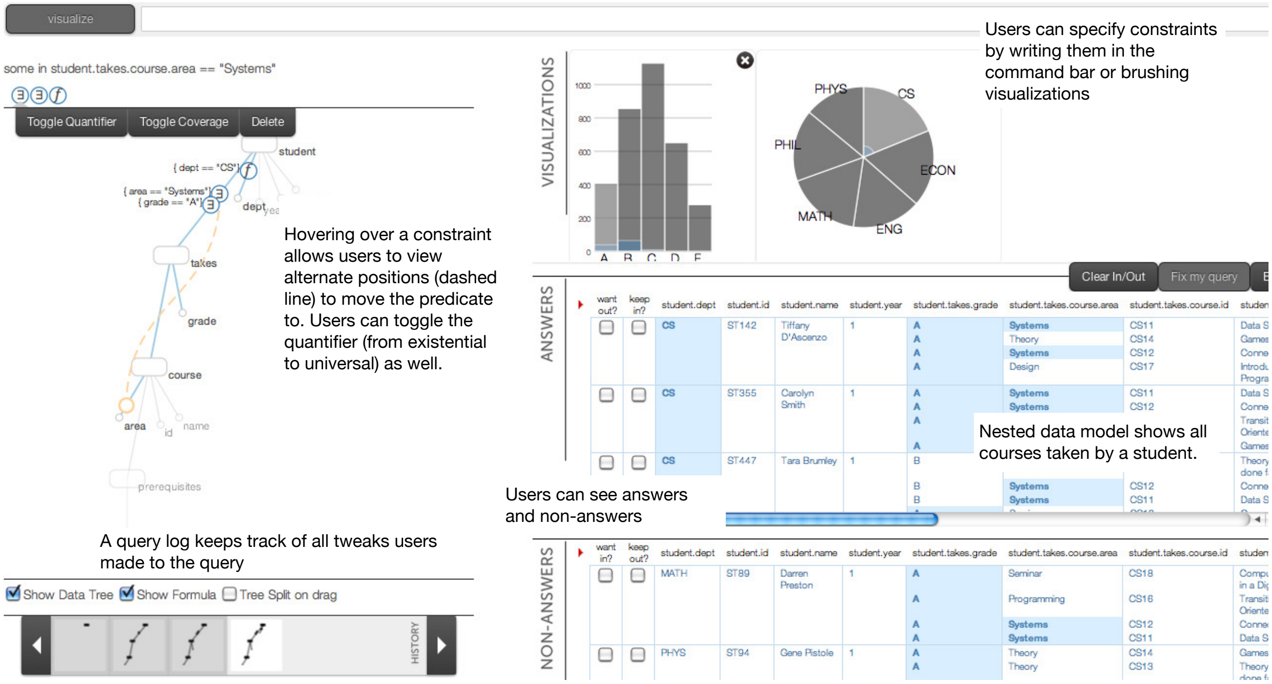


Figure 1: (a) ER-diagram of a school database; (b) The nested-UR schema with student as pivot.

We visually represent nested universal relations using *data trees*. We can transform most relational schema into a nested UR. Figure 1 illustrates such a transformation. At the root of the nested UR is a *pivot* relation around which all other relations are nested. In Figure 1, every `student` has a set of `takes` tuples. The tuples are materialized by taking the join of a `student`'s primary-key with the `takes`' `student` foreign-key.

Similarly every `takes` tuple has a set of `course` tuples, which are materialized by joining the `takes` relation with the `course` relation. Thus, an entire database is converted into a single UR with a hierarchical structure determined by the choice of the pivot relation.



☑ Show Data Tree ☑ Show Formula □ Tree Split on drag

Users can specify constraints by writing them in the command bar or brushing visualizations

Hovering over a constraint allows users to view alternate positions (dashed line) to move the predicate to. Users can toggle the quantifier (from existential to universal) as well.

A query log keeps track of all tweaks users made to the query

Nested data model shows all courses taken by a student.

Users can see answers and non-answers

ANSWERS

want out?	keep in?	student.dept	student.id	student.name	student.year	student.takes.grade	student.takes.course.area	student.takes.course.id	studen
□	□	CS	ST142	Tiffany D'Ascenzo	1	A A A A	Systems Theory Systems Design	CS11 CS14 CS12 CS17	Data S Games Conne Introdu Progra
□	□	CS	ST355	Carolyn Smith	1	A A A A	Systems Systems	CS11 CS12	Data S Conne Transit Oriente Games
□	□	CS	ST447	Tara Brumley	1	B B B	Systems Systems	CS12 CS11	Theory done f Conne Data S

NON-ANSWERS

want in?	keep out?	student.dept	student.id	student.name	student.year	student.takes.grade	student.takes.course.area	student.takes.course.id	studen
□	□	MATH	ST89	Darren Preston	1	A A A A	Seminar Programming Systems Systems	CS18 CS16 CS12 CS11	Compu in a Di Transit Oriente Conne Data S
□	□	PHYS	ST94	Gene Pistole	1	A A	Theory Theory	CS14 CS13	Games Theory done f

Figure 2: DataPlay's query specification interface

Depending on what users are looking for, they can choose different relations as pivots. For example, if we are interested in finding A-students, we pivot the database around `student`. If, however, we are interested in courses with A-students, we pivot the database around `course`.

3.2 Query Model

DataPlay uses a graphical query language: constraint-nodes are superimposed on the data tree's edges to form a *Query Tree*, or a QT. A constraint can only exist on the edges of the path from the pivot to the attribute(s) it evaluates.

Figure 3 illustrates two QTs: one for finding students with at least one A and one for finding straight-A students. Since the nesting hierarchy for a query is predetermined, users do not need to specify how to group tuples for quantification. More importantly, modifying the quantifier type is localized to simply changing the symbol from ∃ to ∀ on the constraint node; the structure of the QT is preserved.

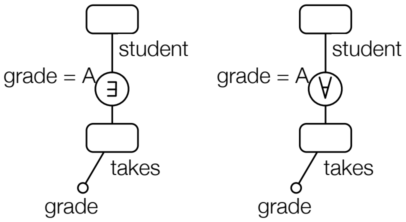


Figure 3: (a) Students with some A's (b) Straight-A's

4. QUERYING WITH DATAPLAY

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

We explain how a user, Jane, will perform each of these steps through an example query task.

Suppose Jane wants to find students who
a) are in the CS department and
b) received A's in *all* Systems courses. We don't care about their grades in other courses.

4.1 Pivoting

Jane connects to the school database, DataPlay visualizes the relationships between the relations (or tables) of the database. She, then, selects one relation, as a *pivot*, for restructuring the database: all relations are joined to the pivot and resulting join-tuples are grouped by the pivot's primary-key into a *nested universal relation* (nested-UR). DataPlay visualizes the schema of the generated nested-UR in a *data tree* and presents a few rows from this relation to help Jane understand the effect of selecting different pivots.

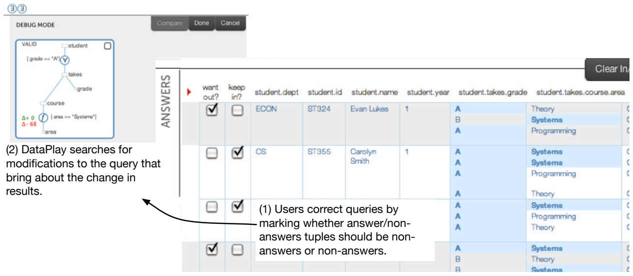
Since Jane is searching for students, she pivots the database around the `student` relation.

4.2 Specifying Constraints

Constraints are simple propositions such as:
`student.dept` = `CS`
`student.takes.grade` = `A` ∨ `student.takes.grade` = `B`

DataPlay allows users to specify constraints by directly writing them into a command-bar or by brushing data-visualizations.

Jane starts by specifying the department constraint. As soon as she writes 'dep' into the command-bar, an auto-



want out?	keep in?	student.dept	student.id	student.name	student.year	student.takes.grade	student.takes.course.area
☑	☐	ECON	ST324	Evan Lukes	1	A B A	Theory Systems Programming
☐	☑	CS	ST355	Carolyn Smith	1	A A A A	Systems Systems Programming Theory
☐	☑					A A A	Systems Programming Theory
☑	☐					A B B	Systems Theory Systems

(2) DataPlay searches for modifications to the query that bring about the change in results.

(1) Users correct queries by marking whether answer/non-answers tuples should be non-answers or non-answers.

Figure 4: DataPlay's query correction tool

suggest feature returns a list of attribute-names that contain 'dep', Jane picks `student.dept`. DataPlay recognizes that it can visualize departments and changes the label of its execute button to 'Visualize'. Clicking 'Visualize' causes a suggestion-box with different visualizations of department data to appear. Jane selects a pie chart of students per department and brushes the CS-pie. In response, DataPlay (i) generates a constraint-node with the propositional formula `student.dept` = `CS`, (ii) adds it on the query tree edge from the `student` node to the `dept` leaf, (iii) executes the query and (iv) updates the answers and non-answers.

Jane goes on to add the course constraint:
`student.takes.course.area` = `Systems`

The command-bar's execute label is now 'Add Constraint'. As before, when Jane clicks it, DataPlay generates a constraint node and adds it to the edge between the `student` and `takes` nodes. Since each student takes a set of courses, DataPlay associates an 'existential' quantifier with the constraint. Finally, Jane adds the grade constraint by brushing an A-bar in a bar chart visualization of grades.

The default query assembled from the constraints finds CS students who took at least one Systems course and got at least one A in any course.

4.3 Semantic Fine-tuning

Syntactic locality implies that each constraint in a query can only have a few localized changes applied to it. This allows DataPlay to visually suggest to users semantic variations to a query by suggesting local changes to a constraint.

Jane wants students who received only A's for all Systems courses. She hovers over the course constraint and DataPlay suggests (Figure 2):

a) toggling the quantifier from existential to universal

b) moving the course constraint underneath the grade constraint such that we only evaluate grades for System's courses

Jane moves the course constraint beneath the grade constraint. She then hovers over the grade constraint to see the range of different tweaks she can apply it. She changes the grade quantifier from existential to universal, now all Systems courses have to have an A.

4.4 Query Correction

Instead of fine-tuning a query by directly manipulating it, Jane can manipulate the results. Jane can mark answer tuples, which should be non-answers, with 'want out' flags. Similarly she can mark non-answers, which should be answers, with 'want in' flags. Jane, thus, revises the results to match the behavior of her intended query. DataPlay searches queries in the neighborhood of the current incorrect query to find one that satisfies her revisions.

This is feasible in DataPlay because (i) syntactic locality guarantees a finite and well-defined search space of query variations — we construct all possible semantic variations to a query by applying local changes to constraints — and (ii) observable complements enable us to present to users tuples missing from their desired answer set, which they would not have seen otherwise.

Figure 4 illustrates the correction interface. Here Jane marks a student with some A's in their Systems courses as a non-answer and the tool suggests a universally-quantified query instead.

5. DEMONSTRATION SCENARIO

The demonstration will consist of two stations. In the first station, we will provide conference attendees with a narrated walkthrough of DataPlay's features. We will demonstrate how to write queries that attendees request. We will also provide details about technical aspects of the system, such as the design of the query engine, the computation of complements or the algorithms used for auto-correcting queries. Attendees can also view a 5-7 minute video tutorial of DataPlay.

In the second station, attendees will be given the chance to use DataPlay to query any of several preloaded databases. Attendees can also participate in a game, where they compete against each other in specifying a query task in either DataPlay or a SQL interface.

6. REFERENCES

[1] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *PODS*, pages 191–200, 1984.

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

[3] D. Maier, J. D. Ullman, and M. Y. Vardi. On the foundations of the universal relation model. *ACM Trans. Database Syst.*, 9(2):283–308, 1984.

[4] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: a database management system for semistructured data. *SIGMOD Rec.*, 26(3):54–66, 1997.

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

[6] P. Reisner, R. F. Boyce, and D. D. Chamberlin. Human factors evaluation of two data base query languages: square and sequel. In *AFIPS*, pages 447–452, 1975.