# Expressive Query Construction through Direct Manipulation of Nested Relational Results

Eirik Bakke
MIT CSAIL
ebakke@csail.mit.edu

David R. Karger
MIT CSAIL
karger@csail.mit.edu

## ABSTRACT

Despite extensive research on visual query systems, the standard way to interact with relational databases remains to be through SQL queries and tailored form interfaces. We consider three requirements to be essential to a successful alternative: (1) query specification through direct manipulation of results, (2) the ability to view and modify any part of the current query without departing from the direct manipulation interface, and (3) SQL-like expressiveness. This paper presents the first visual query system to meet all three requirements in a single design. By directly manipulating nested relational results, and using spreadsheet idioms such as formulas and filters, the user can express a relationally complete set of query operators plus calculation, aggregation, outer joins, sorting, and nesting, while always remaining able to track and modify the state of the complete query. Our prototype gives the user an experience of responsive, incremental query building while pushing all actual query processing to the database layer. We evaluate our system with formative and controlled user studies on 28 spreadsheet users; the controlled study shows our system significantly outperforming Microsoft Access on the System Usability Scale.

## 1. INTRODUCTION

Four decades after Query by Example [61], the broad problem of Making Database Systems Usable [30] remains open. Technical users still interact with relational data through hand-coded SQL, while non-technical users rely on restrictive form- and report-based interfaces tailored, at great cost, for their specific database schema [37, 32, 4]. Queries that involve "complex aggregates, nesting, correlation, and several other features remain on a tall pedestal approachable only by the initiated" [28]. Simple report queries traversing one-to-many relationships in the database schema, such as retrieving "a list of parts, and for each part a list of suppliers and a list of open orders", are painful to define for programmers and largely inaccessible to end users.

Meanwhile, users from a wide range of backgrounds seem happy, indeed eager, to interact with their data if it is served to them in spreadsheet form. "Export to Excel", the joke goes, "is the third most common button in data and business intelligence apps...



*SIGMOD'16 (preprint), June 26-July 01, 2016, San Francisco, CA, USA*



ISBN 978-1-4503-3531-7/16/06.

DOI: http://dx.doi.org/10.1145/2882903.2915210

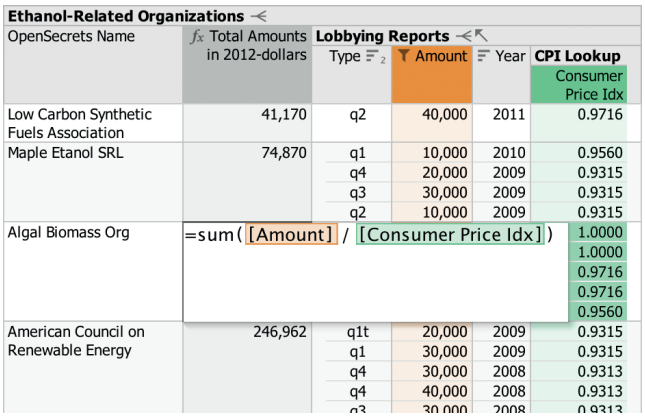



**Figure 1: Spreadsheet-style formula editing in a nested relational result. The query structure is encoded in the table header, which shows three joined table instances (bold labels), one-to-many relationships (≺), sorting (), active filters (, ↖), and a formula ($f_x$).**

after OK and Cancel" [18]. Spreadsheets lack basic database functionality such as joins and views, but demonstrate the great value of usable, general-purpose data manipulation tools [4].

Shneiderman [50] attributes the usability of the spreadsheet to its nature as a *direct manipulation* interface. The properties of such an interface include "visibility of the object of interest", "rapid, reversible, incremental actions", and "replacement of complex command language syntax by direct manipulation of the object of interest". Shneiderman paraphrases Harold Thimbleby: "The display should indicate a complete image of what the current status is, what errors have occurred, and what actions are appropriate."

We agree with Liu and Jagadish [41] that a successful solution to the visual query language problem must come in the form of a spreadsheet-like direct manipulation interface. In particular, we consider three requirements that have yet to be met in a single user interface design:

R1. *Query specification through direct manipulation of results.* The user should build queries incrementally through a sequence of operations performed directly on the data in the database, as seen through the result of each intermediate query [41]. In Shneiderman's terms, the *object of interest* is not the query, but the data, as when working with a spreadsheet.

R2. *The ability to view and modify any part of the current query, including operations performed many steps earlier, without*

*redoing subsequent steps or departing from the direct manipulation interface.* This is tricky in light of R1, because the user will be looking at and manipulating the *result* of a query rather than an actual query expression. The mapping between the two is not obvious. [41]

R3. *SQL-like expressiveness from within the direct manipulation interface.* R1 and R2 can be trivially met if only simple queries are allowed. For example, Excel's *filter* feature works by direct manipulation of results, and allows its complete state to be viewed and modified from within the same interface, but supports only basic selection queries. To compete with SQL, a visual query system should allow the user to express any query commonly supported by SQL implementations, including arbitrary (multi-block) combinations of operations such as joins, calculations, and aggregations.

In this paper, we present SIEUFERD (pronounced *soy-fird*), the first visual query system to meet all of the requirements above in a single user interface design. The key insight is that given a suitable data model for results, the complete structure of a query can be encoded in the schema of the query's own result. This in turn allows the user interface to display the query and its result in a single visual representation, which can then be manipulated directly to modify any part of the query. Specifically, we allow queries to produce results from the nested relational data model [29, 38], and display results using a nested table layout [5]. In our visual representation, the header area of the result's nested table layout encodes the structure of the query, which can then be manipulated using spreadsheet idioms such as formulas and filters. The use of nested results affords a natural visualization of operations such as joins and aggregation, and allows the user to see, in context, intermediate tuples produced in any part of the query.

Using our system, the user can express a relationally complete [17] set of query operators plus calculation, aggregation, outer joins, sorting, and nesting (see Appendix A for details). This covers the full set of query operators generally considered as the minimum to model SQL [6, 26], and expresses, for example, all `SELECT` statements valid in SQL-92. Furthermore, the ability to produce nested results makes our system suitable for complex report creation tasks that would otherwise require multiple SQL queries and custom programming to merge and format results. Our Java-based prototype gives the user an experience of responsive, incremental query building while pushing all query processing to the database layer.

In an initial formative user study, 14 participants were able to solve complex query tasks with a minimal amount of training, with many expressing strong levels of satisfaction with the tool. In a second, controlled study, another 14 participants rated both SIEUFERD and the query designer found in Microsoft Access on the System Usability Scale (SUS) [8] after doing a series of tasks on each. Users rated SIEUFERD 18 points higher on average than Access. This corresponds to a 46 percentage point difference on a percentile scale of other studies in the Business Software category.

## 2. RELATED WORK

Visual query systems have been surveyed by Catarci et al. [11] and, recently, El-Mahgary and Soisalon-Soininen [21]. Systems discussed in this section include, in particular, those that employ direct manipulation, nested results, or optimizations for traversing relationships in the database. We omit systems that rely entirely on text-based languages for query construction. Table 1 categorizes systems by query representation style, and provides an assessment of each system against the requirements set forth in the introduction.

| *Direct Manip.* | *Query Representation* | *Year* | *System* | | *R1* | *R2* | *R3* | *Unrestricted Nested Results* |
|---|---|---|---|---|---|---|---|---|
| Yes | Overlaid on Result | 2014 | GBXT | [2] | X | X | | X |
| | | 2012 | DataPlay | [1] | X | X | | X |
| | | 2006 | Tabulator | [7] | X | X | | X |
| | | 2002 | Polaris/Tableau | [54] | X | X | | |
| | Spreadsheet Formulas | 2016 | Object Spreads. | [43] | X | X | | X |
| | | 2010 | Spreads. as DB | [57] | X | X | | |
| | | 2005 | A1 | [35] | X | X | | X |
| | | 1997 | OOF Spreads. | [16] | X | X | | X |
| | | 1994 | Forms/3 | [9] | X | X | | |
| | Exposed Algebraic | 2013 | Mashroom | [25] | X | | X | X |
| | | 2011 | Wrangler | [34] | X | | X | |
| | | 1991 | TableTalk | [22] | X | | X | X |
| | Hidden Algebraic | 2016 | Gneiss | [13] | X | | X | |
| | | 2013 | GestureDB | [45] | X | | X | |
| | | 2010 | CRIUS | [48] | X | | | X |
| | | 2009 | SheetMusiq | [41] | X | | | |
| | | 2008 | AppForge | [60] | X | | | X |
| | | 1989 | $R^2$ | [27] | X | | X | X |
| No | Diagram-based | 2014 | VisualTPL | [15] | | | | X |
| | | 2009 | App2You | [36] | | | | X |
| | | 2005 | QBB | [47] | | | | |
| | | 2002 | QURSED | [46] | | | | X |
| | | 1990 | QBD | [3] | | | | |
| | Form-based | 2008 | Form Cust. | [33] | | | | |
| | | 1998 | QBEN | [42] | | | | X |
| | | 1997 | ESCHER | [59] | | | | X |
| | | 1989 | PERPLEX | [52] | | | | |
| | | 1977 | QBE | [61] | | | | |

**Table 1: Summary of related systems, evaluated as visual query interfaces. R1 is indicated where some class of queries can be initially specified by direct manipulation of results. R2 is indicated where all parts of such queries can subsequently be modified through similar means. R3 is indicated where the same class of queries is relationally complete and supports aggregation in arbitrary multi-block queries.**

Besides our core requirements, Table 1 also indicates which systems support nested results, i.e. a graphical equivalent of a hierarchical data model such as XML, JSON, or nested relations. This handles report-style queries that encode multiple parallel one-to-many relationships in a single result, as when retrieving "a list of parts, and for each part a list of suppliers and a list of open orders" [5]. Systems that base their result representation on a single flat table of primitive values, such as Tableau [54], are unable to express such queries. The same tends to hold for any system that takes its input from a single joined SQL query, since multivalued dependencies [23] in the flattened result (PARTS↠SUPPLIERS and PARTS↠ORDERS in the preceding example) would interact to produce a pathological number of tuples for even small inputs. Some systems, like Tableau and Gneiss [13], support a restricted form of nesting, where an otherwise flat result table can be grouped into a single-branch hierarchy, or a finite set of such (a *dashboard* in Tableau, or a set of *hierarchical tables* in Gneiss). This still does not handle PARTS↠SUPPLIERS/ORDERS-type queries from the example above. Tableau, as well as other systems based on the pivot table concept, produce cross-tabulated rather than nested results; these concepts are orthogonal. Besides their use in visual query systems, nested data models have been used both in optimization [53, 10] and expressiveness analysis [40] of query languages with aggregate functions.

We first discuss visual query systems that do not fall in the direct manipulation category. *Form-based* systems originated with Query by Example (QBE) [61], where the user populates a set of empty *skeleton tables* with conditions, variables (*examples*), and output

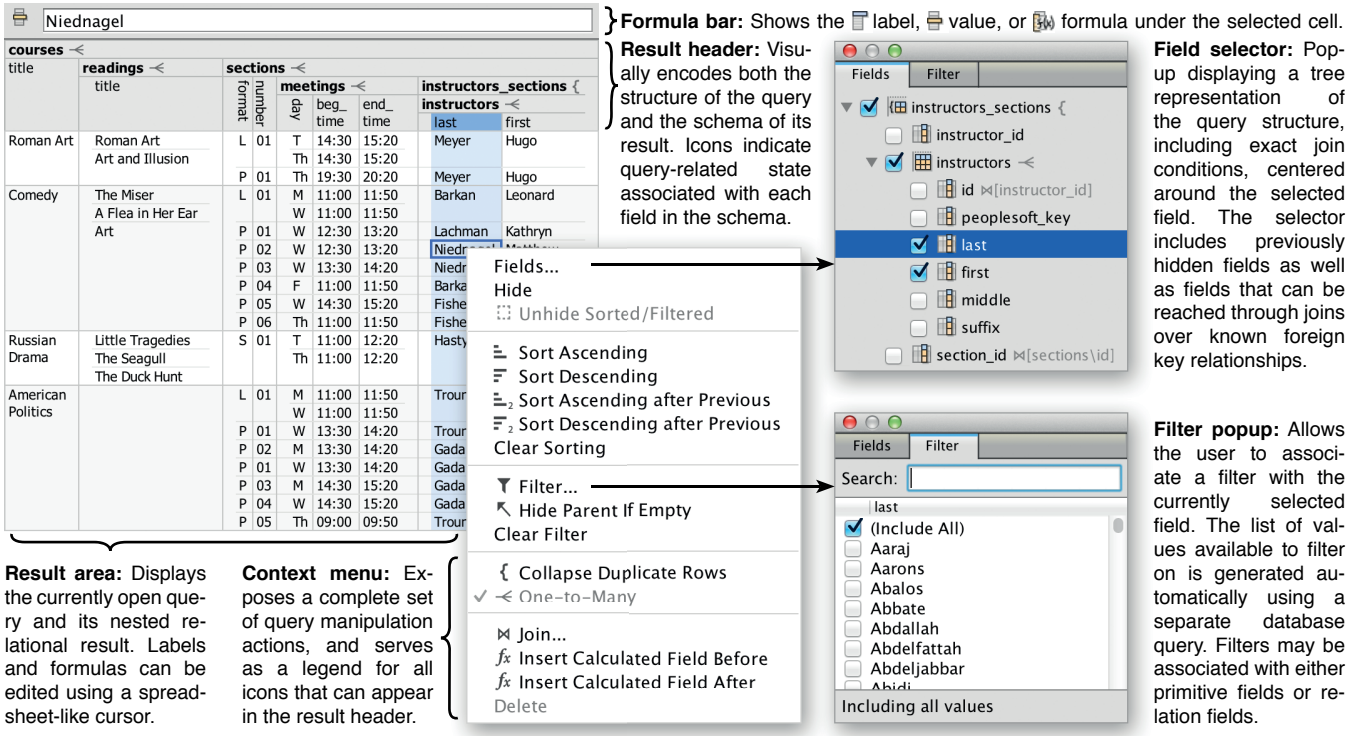



**Figure 2: The SIEUFERD query interface. To create queries, users start from a simple tabular view of a table in the database and add filters, formulas, and nested relations. The integrated result and query representation is displayed continuously as the user interacts with the data. The particular query above instantiates six database tables (one per nested relation), contains five joins (each child relation against its parent), and is evaluated using five generated SQL queries (one for each one-to-many relationship ≺). This query was constructed purely by checking off the appropriate fields and foreign key relationships in the field selector.**

indications. ESCHER [59] and QBEN [42] extend QBE to support nested results, while PERPLEX [52] supports general-purpose logic programming. The ubiquitous search forms of commercial database applications can be seen as restricted versions of QBE tailored for a specific schema; Form Customization [33] generalizes such forms by considering the form designer as part of the query system. In *diagram-based* systems, the user manipulates queries for example through a schema tree or schema diagram, as in Query by Diagram (QBD) [3], Query by Browsing (QBB) [47], QURSED [46], and App2You [36], or through a diagrammatic query plan, as in VisualTPL [15]. The diagram-based query building style is common in commercial tools–Microsoft Access, Navicat, pgAdmin, dbForge, Alteryx etc. The general problem with both form-based and diagram-based interfaces is that users must manipulate queries through an abstract query representation that is divorced from the actual data that is being retrieved. To construct and understand queries, the user must look back and forth between the query representation on one side of the screen and a separate result representation on the other. Thus we do not consider these systems to be direct manipulation interfaces (requirement R1).

In the direct manipulation category, we now consider *algebraic* user interfaces. In such systems, the user builds queries by selecting, one step at a time, a series of operations to be applied to the currently displayed result. Each operation is applied to the result of all previous operations. Formal expressiveness is easy to achieve in algebraic interfaces, since the relevant relational operators can simply be exposed to the user directly. The main problem with algebraic interfaces is that the user has no direct way to, in the words of Liu and Jagadish, "modify an operation specified many steps earlier without redoing the steps afterwards" [41] (requirement R2). For example, in GestureDB [45], the user has no way to modify a filter on a column that was subsequently used in an aggregation or removed with a projection. Similar problems exist in $R^2$ [27], AppForge [60], CRIUS [48], and Gneiss [13]. SheetMusiq [41] provides a partial solution by using an algebra where certain operators can commute out of a complex expression for subsequent modification; however, the technique breaks down for expressions enclosed in binary operators such as joins, set union, or set difference. In other systems, the underlying algebraic expression is exposed directly, as in the procedural *data manipulation scripts* of Wrangler [34], the XQuery-like *mashup scripts* of Mashroom [25], or the diagram-based representation in TableTalk [22]. Thus, only the initial query specification can be done through direct manipulation; tweaking and examination of existing queries must be done with a separate, indirect interface.

With clever use of formulas, Tyszkiewicz [57] shows that existing spreadsheet products can be considered expressive enough to formulate arbitrary SQL queries. If we consider Excel as a query system, however, only a subset of such queries could be said to be constructible by direct manipulation. Heavy reliance on set-based formula functions such as INDEX, MATCH, and SUMPRODUCT means that spreadsheet formulas soon take the role of a text-based query language, with a vocabulary far removed from that of typical query tasks. This would also be the case for spreadsheet programming systems such as Forms/3 [9], Object Oriented Functional Spreadsheets [16], A1 [35], and Object Spreadsheets [43].

Last, we consider direct manipulation systems that *overlay* their query representation on the result of the same query, with the structure of the query reflecting the visual structure of the result. This solves the mapping problem of requirement R2. The problem is

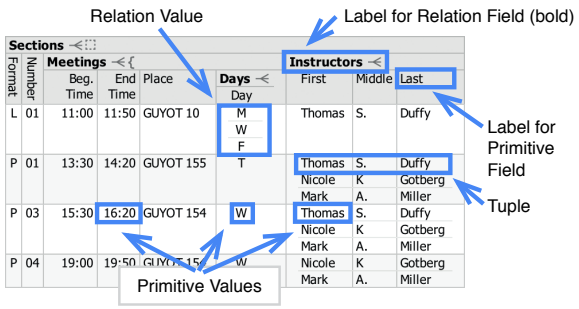



**Figure 3: Terminology of the nested relational data model, illustrated on a nested table layout.**

| | | |
|---|---|---|
| P R | | **Visible.** Whether this field should be visible in the result layout. |
| P R | | **Label.** Presentation label for the field; default is technical column/table name. |
| P R | ⏷ | **Filter.** A filter condition. Filters are stored in a format that can be generated from and restored to a spreadsheet-style filter selection UI. |
| P R | ⫶ | **Sort.** An optional ordinal indicating the position of this field among the parent relation's sort terms, plus an ascending/descending flag. |
| P | ⋈ | **JoinedOn.** An optional reference to a primitive child field of the parent relation's parent relation. This denotes an equijoin condition between this field and the referenced field. |
| P | $f_x$ | **ColumnDefinition.** Either the technical name of a column in the database table specified by InstantiatedTable, or a formula expression over fields in the query model. |
| R | | **InstantiatedTable.** The technical name of a database table to instantiate at this level. Allowed to be absent, in which case semantics are equivalent to instantiating a single-tuple, zero-column table. |
| R | { | **CollapseDuplicateRows.** False by default, in which case the primary key fields of InstantiatedTable are projected in intermediate and retrieved results even if not Visible. |
| R | ↖ | **HideParentIfEmpty.** If true, an inner join is used between this relation and its parent. Set automatically by the filter UI, but can be overridden. |

**Table 2: Properties in the SIEUFERD query model, associated with each field in the nested relational schema that defines a visual query. P, R, and P R indicate properties applicable to primitive fields, relation fields, or both, respectively. Properties with icons correspond directly to icons shown in the result area and actions in the user-accessible context menu from Figure 2.**

that current such representations are not expressive enough to support arbitrary queries (requirement R3). For example, the direct manipulation interfaces of Tabulator [7] and GBXT [2] support filters and joins over schema relationships, but are unable to express calculation, aggregation, general-purpose joins, or other binary operators. In DataPlay [1], direct manipulation is used only to choose between universal and existential qualifiers. Tableau [54] allows a large class of two-dimensional visualizations to be created and manipulated through direct manipulation of table headers and corresponding axis *shelves*; however, queries involving calculations or binary operators must be configured using a separate interface rather than through direct manipulation. Our own system is the first to achieve SQL-like expressiveness from within a direct manipulation interface based on an overlaid query/result representation.

# 3. SYSTEM DESCRIPTION

## 3.1 Overview

Our core query building interface is shown in Figure 2. All user interactions are initiated from the *result area*, which shows the current query's nested relational result, formatted using a nested table layout. In a nested table layout, the table's *header* area visually encodes the schema of the nested result, including which fields are nested under others in the hierarchical schema. Because our system maps all query-related state to specific fields in the result schema, the result's table header simultaneously becomes a visual representation of the query that generated it. A set of icons, carefully designed to allow every aspect of the query state to be represented in the header, is used to augment the information that can be derived from the names and positions of fields.

Starting from any selection of fields (columns) in the result area, the user may open a *context menu* of query-related actions, which also serves as a legend for icons that may appear in the result header. Query actions modify the query state, not the data in the database. Whenever a visual query is modified, the system generates and executes one or more corresponding SQL queries to evaluate it, merges the returned flat results into a single nested result, and displays the latter to the user. At the same time, the fields and iconography in the new result's header reflect the updated state of the modified query.

To keep the result layout compact, several aspects of the query state are indicated with icons in the header but are not displayed in full until the user requests it. In these cases we leverage well-established spreadsheet idioms to expose the underlying state. A filter icon (⏷) next to a field label indicates the presence of a filter on that field, which can be manipulated by opening the *filter popup* from the context menu. A formula icon ($f_x$) indicates that the primitive field in question is a calculated field with an associated spreadsheet-style formula. The actual formula can be edited using the *formula bar* above the result area, or directly in any non-header cell belonging to the field's column, as illustrated in Figure 1. Finally, as in a spreadsheet, our system allows fields (columns) to be hidden from view and later recalled for inspection. If the hidden field was used for filtering or sorting, or is referenced from a formula, a dashed cell icon (⁝) is shown for the relevant dependent field to indicate that the visible result depends on a hidden portion of the query. Hidden fields can be recalled using the *field selector* popup, which shows an expandable list of available fields, centered around the field it was opened for. The field selector also serves to suggest new joins over known foreign key relationships, modeled as pre-existing hidden fields, and to display exact join conditions.

For the remainder of this paper, we will use the following terminology when referring to concepts in the nested relational data model: A *value* is either a *primitive* or a *relation*, where a relation is defined as a set of *tuples*, each containing a set of *fields* identified by *labels*, each containing a value, recursively. The *schema* of a value either defines the value to be a primitive, or defines the value to be a relation, with schemas further specified for each of the latter's fields, recursively. See Figure 3.

## 3.2 Query Model

We now discuss the specific structure of queries in our system. A visual query is modeled as a nested relational schema that has been annotated with query- and presentation-related properties on each field. See Table 2. We refer to the annotated schema as the SIEUFERD *query model*. When SQL queries are generated from a visual query and flat result sets have been assembled into a nested relational result, the schema of the nested result is identical to the schema in the query model. This correspondence makes it straightforward to translate high-level user interactions on the visualized query result to concrete modifications on the underlying query model, and conversely, to indicate the state of the query model in the table header of the visualized result.

**Table instantiation.** As a basic rule, each relation in the query model gets to retrieve data from one concrete table in the underlying database; that relation is said to *instantiate* the database table.

The following is a simple query that instantiates the table called COURSES and displays a selection of its fields:

| courses ≺ | | | | | |
|---|---|---|---|---|---|
| id | area_id | title | may_pdf | may_audit | exam_type |
| 56 | 2 | Roman Art | N | Y | Other |
| 177 | 2 | Comedy | Y | Y | Final |
| 845 | 2 | Russian Drama | N | N | Other |
| 1795 | 4 | American Politics | Y | Y | Final |
| 2566 | | Junior Seminars | N | N | Other |
| 3921 | 4 | Judicial Politics | Y | Y | Final |

In the SIEUFERD query model, the query above is represented as a nested relational schema whose root relation references the COURSES table from its INSTANTIATEDTABLE property, with primitive child fields storing the technical name of each table column in their respective COLUMNDEFINITION properties. Similar encodings are used for all query states that will be discussed in this section; see Table 2 for details.

**Nesting and joins.** Queries need to be able to incorporate data from multiple tables. Commonly, tables need to be equijoined together, for example when the user wishes to examine data spread across foreign key relationships in a normalized database schema. In the SIEUFERD query model, the introduction of a new table instance can be done by defining a *nested relation*, optionally constrained by an equijoin condition against its parent relation:

| courses ≺ | | | | | | readings ≺ | | | |
|---|---|---|---|---|---|---|---|---|---|
| id | area_id | title | may_pdf | may_audit | exam_type | id | ⋈ course_id | author_name | title |
| 56 | 2 | Roman Art | N | Y | Other | 44 | 56 | Ramage | Roman Art |
| | | | | | | 8,838 | 56 | Gombrich | Art and Illusion |
| 177 | 2 | Comedy | Y | Y | Final | 4,998 | 177 | Moliere | The Miser |
| | | | | | | 12,138 | 177 | Feydeau | A Flea in Her Ear |
| | | | | | | 16,878 | 177 | Reza | Art |
| 845 | 2 | Russian Drama | N | N | Other | 603 | 845 | Pushkin | Little Tragedies |
| | | | | | | 9,207 | 845 | Chekhov | The Seagull |
| | | | | | | 12,366 | 845 | Vampilov | The Duck Hunt |
| 1795 | 4 | American Politics | Y | Y | Final | | | | |
| 2566 | | Junior Seminars | N | N | Other | 9,935 | 2566 | Pierre Loti | India |
| 3921 | 4 | Judicial Politics | Y | Y | Final | 2,570 | 3921 | Rosenberg, Gerald | The Hollow Hope |
| | | | | | | 17,629 | 3921 | Lazarus, Edward | Closed Chambers |

In the query above, the nested relation READINGS instantiates the database table with the same name, and equijoins itself against its parent relation COURSES on the COURSE_ID field, as indicated by the join icon (⋈) on the latter. The other side of the equijoin condition is the ID field in the COURSES relation. The latter information is omitted from the result layout to save space, but is displayed in the field selector (Figure 2). The one-to-many icon (≺) on the READINGS relation indicates that our system decided the latter may contain more than one tuple for each corresponding tuple in COURSES, the parent relation.

The joins described here have different semantics than the traditional flat joins encountered in SQL and most other visual query tools. Rather than duplicating tuples on one side of the operator for each occurrence of a matching tuple on the other, each tuple from the parent side of the join has a nested relation added to it holding zero or more matching tuples from the child side. This operator is known formally as a *nest equijoin* [53], though we will simply use the term *join* when unambiguous. One convenient property of nest equijoins is that tuples on the left-hand side of the operator do not disappear when the join fails to find matching tuples on the right; this can be seen in the query above for the course AMERICAN POLITICS, which has no books in its reading list.

It is often desirable to hide technical primary key fields, fields made redundant by equijoin conditions (e.g. COURSE_ID), or otherwise uninteresting fields, for presentation purposes. Continuing the example above, our query model allows us to hide several fields without altering the query semantics:

| courses ≺ | | | | | |
|---|---|---|---|---|---|
| title | may_pdf | may_audit | exam_type | readings ≺ author_name | readings ≺ title |
| Roman Art | N | Y | Other | Ramage | Roman Art |
| | | | | Gombrich | Art and Illusion |
| Comedy | Y | Y | Final | Moliere | The Miser |
| | | | | Feydeau | A Flea in Her Ear |
| | | | | Reza | Art |
| Russian Drama | N | N | Other | Pushkin | Little Tragedies |
| | | | | Chekhov | The Seagull |
| | | | | Vampilov | The Duck Hunt |
| American Politics | Y | Y | Final | | |
| Junior Seminars | N | N | Other | Pierre Loti | India |
| Judicial Politics | Y | Y | Final | Rosenberg, Gerald | The Hollow Hope |
| | | | | Lazarus, Edward | Closed Chambers |

The hidden fields could be recalled at any time using the field selector. As before, the field selector can also be used to see the exact join conditions between READINGS and COURSES.

Nested relations can be used very effectively to display data spread over many tables in a database schema. In the following example, we pull data from five database tables to see more information about each university course:

| courses ≺ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| area | | title | readings ≺ | | sections ≺ | | | | | |
| title | code | | author_name | title | type | num | meetings ≺ day | start | end |
| Literature and the Arts | LA | Roman Art | Ramage | Roman Art | L | 01 | T | 14:30 | 15:20 |
| | | | Gombrich | Art and Illusion | | | Th | 14:30 | 15:20 |
| | | | | | P | 01 | Th | 19:30 | 20:20 |
| Literature and the Arts | LA | Comedy | Moliere | The Miser | L | 01 | M | 11:00 | 11:50 |
| | | | Feydeau | A Flea in Her Ear | | | W | 11:00 | 11:50 |
| | | | | | P | 01 | W | 12:30 | 13:20 |
| | | | Reza | Art | P | 02 | W | 12:30 | 13:20 |
| | | | | | P | 03 | W | 13:30 | 14:20 |
| | | | | | P | 04 | F | 11:00 | 11:50 |
| | | | | | P | 05 | W | 14:30 | 15:20 |
| | | | | | P | 06 | Th | 11:00 | 11:50 |
| Literature and the Arts | LA | Russian Drama | Pushkin | Little Tragedies | S | 01 | T | 11:00 | 12:20 |
| | | | Chekhov | The Seagull | | | Th | 11:00 | 12:20 |
| | | | Vampilov | The Duck Hunt | | | | | |

Notice that tuples in the READINGS relation occur independently of tuples in the SECTIONS relation; this kind of visualization can not be constructed in tools based on flat tabular results (see Related Work). Also notice the absence of the of the one-to-many icon (≺) on the AREA relation: because the latter relation was joined on its instantiated table's primary key, our system deduced that at most one tuple can exist in AREA for each parent tuple in COURSES.

**Sorting.** Each nested relation can be sorted on a sequence of its direct child fields, indicated by subscripted sort icons ($\text{sort}_{123}$) on the latter. In the following example, the root-level COURSES relation is sorted ascending on the MAX_ENROLL field, while individual sets of READINGS are sorted by AUTHOR_NAME, then by TITLE:

| courses ≺ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| title | max_enroll | readings ≺ author_name | readings ≺ title | sections ≺ type | num | meetings ≺ day | start | end |
| Russian Drama | 0 | Chekhov | The Seagull | S | 01 | T | 11:00 | 12:20 |
| | | Pushkin | Little Tragedies | | | Th | 11:00 | 12:20 |
| | | Vampilov | The Duck Hunt | | | | | |
| Junior Seminars | 12 | Pierre Loti | India | S | 01 | M | 13:30 | 16:20 |
| Judicial Politics | 24 | Lazarus, Edward | Closed Chambers | L | 01 | W | 11:00 | 11:50 |
| | | | | | | F | 11:00 | 11:50 |
| | | Rosenberg, Gerald | The Hollow Hope | P | 01 | Th | 14:30 | 15:20 |
| | | | | P | 02 | W | 13:30 | 14:20 |
| | | | | P | 04 | W | 11:00 | 11:50 |
| | | | | P | 03 | T | 11:00 | 11:50 |
| Roman Art | 25 | Gombrich | Art and Illusion | L | 01 | T | 14:30 | 15:20 |
| | | Ramage | Roman Art | | | Th | 14:30 | 15:20 |
| | | | | P | 01 | Th | 19:30 | 20:20 |

It is possible to sort on both primitive and relation fields, though we omit the exact semantics of the latter case here. Following any explicit sort terms, our system automatically sorts every relation on a tuple-identifying subset of its retrieved fields. This ensures that all query results are retrieved in a deterministic order. The automatic sort is usually on an indexed primary key; see *set projection* below.

**Filter.** Using the filter popup (Figure 2), a filter can be defined on any field, indicated by the filter icon (▼). Filters on relation fields restrict the set of tuples retrieved in that relation, while filters

on primitive fields restrict the tuples of the parent relation. In the following example, the MEETINGS relation is filtered to show only tuples for which the DAY is W:

**courses**

| title | max _enroll | readings: author_name | readings: title | sections: type | sections: num | meetings: day | meetings: start | meetings: end |
|---|---|---|---|---|---|---|---|---|
| Comedy | 99 | Moliere | The Miser | L | 01 | W | 11:00 | 11:50 |
| | | Feydeau | A Flea in Her Ear | P | 01 | W | 12:30 | 13:20 |
| | | | | P | 02 | W | 12:30 | 13:20 |
| | | Reza | Art | P | 03 | W | 13:30 | 14:20 |
| | | | | P | 05 | W | 14:30 | 15:20 |
| American Politics | 78 | | | L | 01 | W | 11:00 | 11:50 |
| | | | | P | 01 | W | 13:30 | 14:20 |
| | | | | P | 01 | W | 13:30 | 14:20 |
| | | | | P | 04 | W | 14:30 | 15:20 |
| Judicial Politics | 24 | Rosenberg, Gerald | The Hollow Hope | L | 01 | W | 11:00 | 11:50 |
| | | | | P | 02 | W | 13:30 | 14:20 |
| | | Lazarus | Closed | P | 04 | W | 11:00 | 11:50 |

By default, the effect of a filter in a nested relation is propagated all the way to the root of the query by means of a HIDE PARENT IF EMPTY setting on each intermediate relation, indicated by the arrow-towards-root icon (↖) on the SECTIONS and MEETINGS relations above. In the example, the courses ROMAN ART and RUSSIAN DRAMA have disappeared because they do not have any Wednesday sections. If, rather than retrieving "a list of courses with at least one Wednesday section", we wanted to retrieve "a list of all courses, showing sections on Wednesday only", we could deactivate HIDE PARENT IF EMPTY on the SECTIONS relation:

**courses**

| title | max _enroll | readings: author_name | readings: title | sections: type | sections: num | meetings: day | meetings: start | meetings: end |
|---|---|---|---|---|---|---|---|---|
| Roman Art | 25 | Ramage | Roman Art | | | | | |
| | | Gombrich | Art and Illusion | | | | | |
| Comedy | 99 | Moliere | The Miser | L | 01 | W | 11:00 | 11:50 |
| | | Feydeau | A Flea in Her Ear | P | 01 | W | 12:30 | 13:20 |
| | | | | P | 02 | W | 12:30 | 13:20 |
| | | Reza | Art | P | 03 | W | 13:30 | 14:20 |
| | | | | P | 05 | W | 14:30 | 15:20 |
| Russian Drama | 0 | Pushkin | Little Tragedies | | | | | |
| | | Chekhov | The Seagull | | | | | |
| | | Vampilov | The Duck Hunt | | | | | |
| American Politics | 78 | | | L | 01 | W | 11:00 | 11:50 |
| | | | | P | 01 | W | 13:30 | 14:20 |
| | | | | P | 01 | W | 13:30 | 14:20 |
| | | | | P | 04 | W | 14:30 | 15:20 |
| Junior Seminars | 12 | Pierre Loti | India | | | | | |
| Judicial Politics | 24 | Rosenberg | The Hollow | L | 01 | W | 11:00 | 11:50 |

**Formulas.** An important part of the expressiveness offered by SQL is the ability to include scalar and aggregate computations over primitive values in any part of the query. In the SIEUFERD query model, both kinds of calculations are supported by means of *calculated fields*. A calculated field is a primitive field, added to any relation by the user, that takes its value from a *formula* rather than from a particular column in an instantiated database table. Like other fields, calculated fields can be sorted or filtered on.

SIEUFERD formulas are syntactically similar to spreadsheet formulas, but belong to and reference entire columns of field values rather than hard-coded ranges of cells. This allows SIEUFERD queries, like SQL queries, to be defined independently of the exact data that might reside in a database at any given time. Without this design, the user might have to rewrite formulas if the data in the underlying data source changes, or if other parts of the query are changed in such a way as to add or remove tuples in the result. Forgetting to update formulas when input data is changed is a common kind of error in spreadsheets [31, 12], which we avoid.

The restriction that calculated fields always be primitive fields is an important one; we do not wish formulas to take the role of a textual query language embedded within the visual one. Formulas do not provide a relational algebra, but rather allow simple computations over primitive values.

Continuing the course catalog example, we can calculate the duration of each meeting of a course section:

**courses**

| title | sections: type | sections: num | meetings: day | meetings: start | meetings: end | $f_x$ duration |
|---|---|---|---|---|---|---|
| Roman Art | L | 01 | T | 14:30 | 15:20 | 50 |
| | | | Th | 14:30 | 15:20 | =minutes([end] - [start]) |
| | P | 01 | Th | 19:30 | 20:20 | 50 |
| Comedy | L | 01 | M | 11:00 | 11:50 | 50 |
| | | | W | 11:00 | 11:50 | 50 |
| | P | 01 | W | 12:30 | 13:20 | 50 |
| | P | 02 | W | 12:30 | 13:20 | 50 |
| | P | 03 | W | 13:30 | 14:20 | 50 |
| | P | 04 | F | 11:00 | 11:50 | 50 |
| | P | 05 | W | 14:30 | 15:20 | 50 |
| | P | 06 | Th | 11:00 | 11:50 | 50 |
| Russian Drama | S | 01 | T | 11:00 | 12:20 | 80 |
| | | | Th | 11:00 | 12:20 | 80 |

The calculated field DURATION, marked with the formula icon ($f_x$), is evaluated once for each tuple in MEETINGS, its containing relation. Using another calculated field, we can add up the durations as well, at the level of each course:

=sum([duration])

**courses**

| title | $f_x$ total duration | sections: type | sections: num | meetings: day | meetings: start | meetings: end | $f_x$ duration |
|---|---|---|---|---|---|---|---|
| Roman Art | 150 | L | 01 | T | 14:30 | 15:20 | 50 |
| | | | | Th | 14:30 | 15:20 | 50 |
| | | P | 01 | Th | 19:30 | 20:20 | 50 |
| Comedy | 400 | L | 01 | M | 11:00 | 11:50 | 50 |
| | | | | W | 11:00 | 11:50 | 50 |
| | | P | 01 | W | 12:30 | 13:20 | 50 |
| | | P | 02 | W | 12:30 | 13:20 | 50 |
| | | P | 03 | W | 13:30 | 14:20 | 50 |
| | | P | 04 | F | 11:00 | 11:50 | 50 |
| | | P | 05 | W | 14:30 | 15:20 | 50 |
| | | P | 06 | Th | 11:00 | 11:50 | 50 |
| Russian Drama | 160 | S | 01 | T | 11:00 | 12:20 | 80 |
| | | | | Th | 11:00 | 12:20 | 80 |

When using aggregate functions such as SUM or COUNT, the relation in which the calculated field is defined determines the level at which aggregate values are grouped. In the example above, because the TOTAL DURATION field is a child of the COURSES relation, a total is calculated for each course rather than, say, for each section. Each course includes in its total only tuples from the MEETINGS relation that are descendants of that course's tuple in the COURSES relation.

It is permitted for a formula to reference fields outside its own containing relation, as in the following example:

=100 * [duration] / [total duration]

**courses**

| title | $f_x$ total duration | sections: type | sections: num | meetings: day | meetings: start | meetings: end | $f_x$ duration | $f_x$ percent |
|---|---|---|---|---|---|---|---|---|
| Roman Art | 150 | L | 01 | T | 14:30 | 15:20 | 50 | 33.33 |
| | | | | Th | 14:30 | 15:20 | 50 | 33.33 |
| | | P | 01 | Th | 19:30 | 20:20 | 50 | 33.33 |
| Comedy | 400 | L | 01 | M | 11:00 | 11:50 | 50 | 12.50 |
| | | | | W | 11:00 | 11:50 | 50 | 12.50 |
| | | P | 01 | W | 12:30 | 13:20 | 50 | 12.50 |
| | | P | 02 | W | 12:30 | 13:20 | 50 | 12.50 |
| | | P | 03 | W | 13:30 | 14:20 | 50 | 12.50 |
| | | P | 04 | F | 11:00 | 11:50 | 50 | 12.50 |
| | | P | 05 | W | 14:30 | 15:20 | 50 | 12.50 |
| | | P | 06 | Th | 11:00 | 11:50 | 50 | 12.50 |
| Russian Drama | 160 | S | 01 | T | 11:00 | 12:20 | 80 | 50.00 |
| | | | | Th | 11:00 | 12:20 | 80 | 50.00 |

Here, the formula in the PERCENT field references the TOTAL DURATION field of the outer COURSES relation. This is analogous to a correlated subquery in SQL. Such *outward* references are not crucial to our query model's expressiveness; we eliminate them using a decorrelation technique like that described by Van den Bussche and Vansummeren [58, p. 8].

**Filters and aggregate functions.** When an aggregate function references a relation with a filter applied to it, the filter is evaluated

before the aggregate. In the following example, the SECTIONS relation is filtered to only include lecture-type sections. The TOTAL DURATION for each course changes accordingly:

**courses**

| title | $f_x$ total duration | sections | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | type | num | meetings | | | | |
| | | | | day | start | end | $f_x$ duration | $f_x$ percent |
| Roman Art | 100 | L | 01 | T | 14:30 | 15:20 | 50 | 50.00 |
| | | | | Th | 14:30 | 15:20 | 50 | 50.00 |
| Comedy | 100 | L | 01 | M | 11:00 | 11:50 | 50 | 50.00 |
| | | | | W | 11:00 | 11:50 | 50 | 50.00 |
| Russian Drama | 0 | | | | | | | |
| American Politics | 100 | L | 01 | M | 11:00 | 11:50 | 50 | 50.00 |
| | | | | W | 11:00 | 11:50 | 50 | 50.00 |
| Junior Seminars | 0 | | | | | | | |
| Judicial Politics | 100 | L | 01 | W | 11:00 | 11:50 | 50 | 50.00 |
| | | | | F | 11:00 | 11:50 | 50 | 50.00 |

It is equally valid to define a filter on the output side of an aggregate, e.g. on TITLE or TOTAL DURATION in the example above.

**Flat joins.** Traditional flat joins can be expressed by referencing a descendant relation from a formula without enclosing the reference in an aggregate function. In the following example, each course title is repeated once for each distinct author name in the reading list, because the AUTHOR REFERENCE field in the COURSES relation references the READINGS relation without the use of an aggregate function:

**courses**

| title | exam_ type | author reference $f_x$ | readings author_name | title |
|---|---|---|---|---|
| Roman Art | Other | Gombrich | Gombrich | Art and Illusion |
| Roman Art | Other | Ramage | Ramage | Roman Art |
| Comedy | Final | Feydeau | Feydeau | A Flea in Her Ear |
| Comedy | Final | =[author_name] | | The Miser |
| Comedy | Final | Reza | Reza | Art |
| Russian Drama | Other | Chekhov | Chekhov | The Seagull |
| Russian Drama | Other | Pushkin | Pushkin | Little Tragedies |
| Russian Drama | Other | Vampilov | Vampilov | The Duck Hunt |
| American Politics | Final | | | |
| Junior Seminars | Other | Pierre Loti | Pierre Loti | India |
| Judicial | Final | Lazarus | Lazarus, Edward | Closed Chambers |

The actual behavior is that of a left join, with a null value being returned for the course AMERICAN POLITICS, which has no readings in its reading list. To express an inner join instead, the HIDE PARENT IF EMPTY setting could be enabled on the READINGS relation. The left join semantics of these *inward* formula references help our visual query language maintain some desirable properties. In particular, the mere introduction of a new calculated field (e.g. AUTHOR REFERENCE) will never cause tuples to disappear from said field's containing relation (COURSES).

**Set projection.** By default, tuples retrieved for a relation always include the primary key fields of the relation's instantiated table, even if the user has hidden those fields from view. This allows our system to keep result tuples in a stable order as the user hides or shows fields, and to keep a one-to-one relationship between tuples on the screen and tuples in instantiated database tables. It also allows us to generate more efficient SQL queries, for example by avoiding expensive SELECT DISTINCT statements. The automatic inclusion of primary key fields in the projection of a particular relation can be avoided by means of the HIDE DUPLICATE ROWS option, indicated by the bracket icon ({):

**courses**

| title | sections type | status |
|---|---|---|
| Roman Art | L | O |
| | P | X |
| Comedy | L | O |
| | P | O |
| | P | O |
| | P | O |
| | P | O |
| | P | X |
| | P | X |
| Russian Drama | S | O |

**courses**

| title | sections type | status |
|---|---|---|
| Roman Art | L | O |
| | P | X |
| Comedy | L | O |
| | P | O |
| | P | X |
| Russian Drama | S | O |
| American Politics | L | O |
| | P | X |
| Junior Seminars | S | O |
| Judicial Politics | L | O |

## 3.3 Query Building

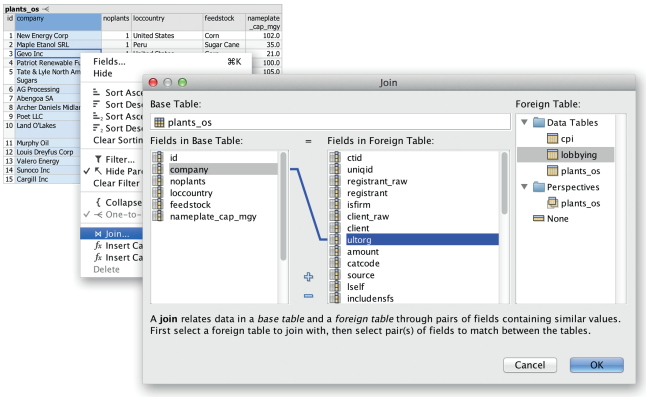
Having explained the query model, we now show how the user would actually build queries using our direct manipulation interface. We do this by means of an example query building session. The user is an investigative journalist who is writing a story about ethanol biofuel lobbying [20]. She has compiled, in the table PLANTS_OS, a list of major ethanol producers[1], and would like to find the total lobbying expenditures of each. Another table, LOBBYING, contains quarterly lobbying reports from US corporations in the years 1998 through 2012 (727,927 tuples)[2].

**Base table.** The user starts by opening the table of ethanol producers as a template for the new query:

**plants_os**

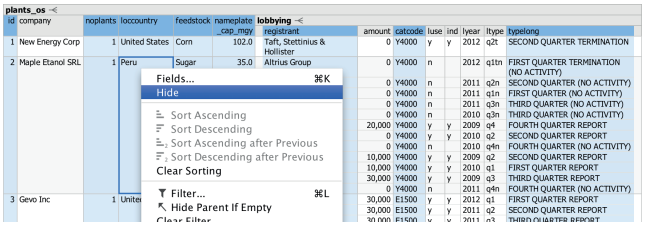
| id | company | noplants | loccountry | feedstock | nameplate_cap_mgy |
|---|---|---|---|---|---|
| 1 | New Energy Corp | 1 | United States | Corn | 102.0 |
| 2 | Maple Etanol SRL | 1 | Peru | Sugar Cane | 35.0 |
| 3 | Gevo Inc | 1 | United States | Corn | 21.0 |
| 4 | Patriot Renewable Fuels | 1 | United States | Corn | 100.0 |
| 5 | Tate & Lyle North American Sugars | 1 | United States | Corn | 105.0 |
| 6 | AG Processing | 1 | United States | Corn | 52.0 |

**Join.** To add another table to the query, the user selects the column or columns to join on and invokes the JOIN action from the context menu. This opens a dialog box for selecting the table to join with, in this case LOBBYING, and for selecting the corresponding columns from the latter to be matched in an equijoin constraint. The user joins the PLANTS_OS and LOBBYING tables on the COMPANY and ULTORG fields, respectively:



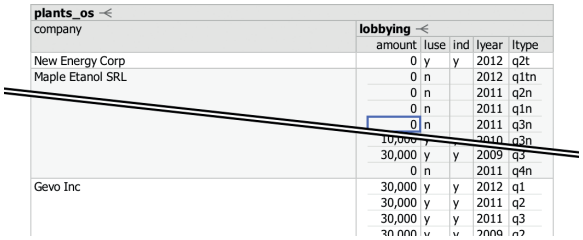

In cases where the database defines explicit foreign key relationships between tables, use of the above JOIN dialog is unnecessary; instead, all available joins will be available as hidden relations in the field selector. The effect is a schema navigation capability analogous to that of QBB [47], AppForge [60], and App2You [36].

**Hide fields.** After the join, a lot of columns are shown, so the user selects a few of them and invokes the HIDE action:



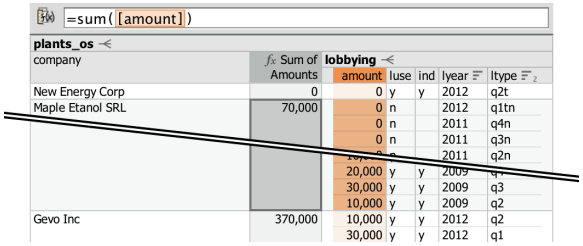

It is now easier to get a sense of the data. We have a new child relation field, called LOBBYING, containing the lobbying reports for each company:

[1]Renewable Fuels Association/Maple Etanol SRL (2012)
[2]The Center for Responsive Politics (2012) https://www.opensecrets.org



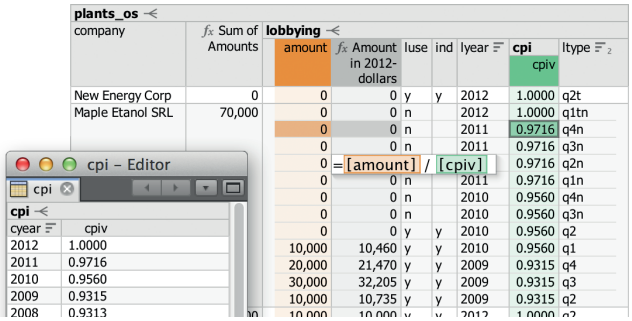

**Sort.** The user decides to sort the lobbying reports for each company most-recent-first, invoking the SORT DESCENDING action on the LYEAR field and then invoking the SORT DESCENDING AFTER PREVIOUS action on the LTYPE field. This sorts individual LOBBYING relations by year ($\blacktriangledown$) and then by quarter ($\blacktriangledown_2$):


**Aggregate formula.** The user would now like to calculate a total lobbying amount for each company. She invokes the INSERT CALCULATED FIELD AFTER action to insert a calculated field ($fx$) next to the COMPANY field, and enters the name SUM OF AMOUNTS in the new column's label cell. She then moves the cursor to one of the column's value cells, and enters a sum formula, clicking the AMOUNT column to insert the column reference:

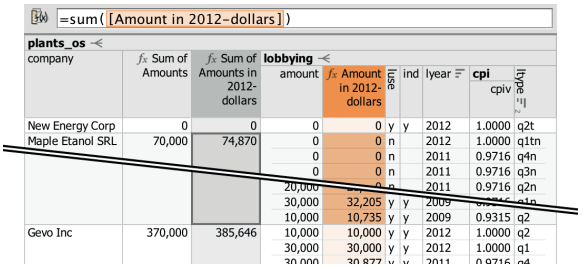



Unlike in a spreadsheet, there is no need to "drag down" the sum formula; it is always evaluated once for each tuple in PLANTS_OS, its parent relation.

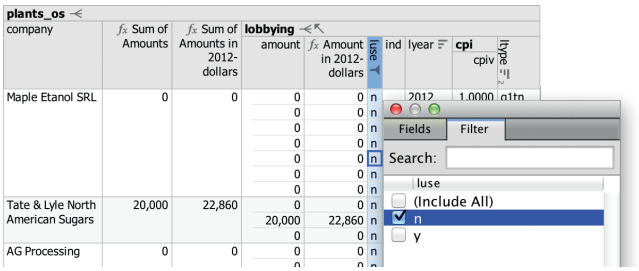
**Scalar formula.** Reported lobbying amounts come from different years, some going back to 1998. The user would like to calculate inflation-corrected totals. A separate table CPI contains yearly Consumer Price Index values normalized for 2012. The user performs another JOIN, this time between LOBBYING and CPI, on the LYEAR and CYEAR fields, respectively. This brings the CPIV value for each lobbying report's year into the nested result. The user then adds another calculated field, this time under the same relation as the existing AMOUNT field, and enters a formula that calculates the inflation-adjusted amount for each report. We here have a useful example of an inward formula reference (to CPIV) that is not enclosed in an aggregate function:




A new inflation-adjusted total can now be added as a calculated field at the PLANTS_OS level, shown adjacent to the existing non-adjusted sum:

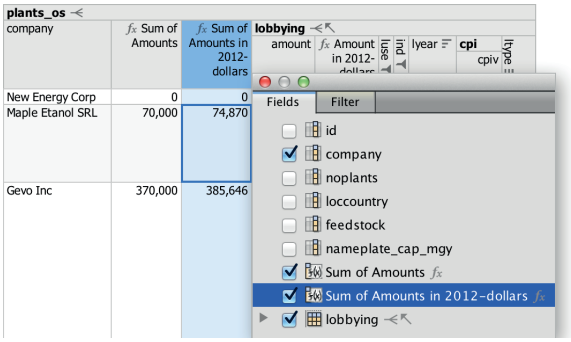



**Filter.** Lobbying reports may sometimes be amended, in which case the superseded reports should be excluded from totals to avoid double counting. The user can look for superseded reports by invoking the FILTER action on the LUSE field and selecting the value N:

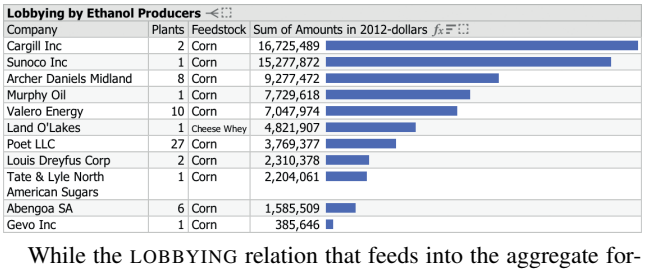



The user sees that there are superseded reports in the database with non-zero dollar amounts, and inverts the filter to exclude them.

**Select fields.** The user now decides to hide the individual reports altogether and instead reintroduce some of the fields that were hidden from the PLANTS_OS relation before, using the field selector:




**Final touches.** The user edits the field labels to make them a bit more readable, and sorts the companies by their lobbying totals. The underlying SQL column names can still be seen in the field selector. The user also enables a formatting option on the last column to produce a bar chart visualization. The result now looks presentable:




While the LOBBYING relation that feeds into the aggregate formula is now hidden, the user could easily make it visible again from

the field selector, like she did for the previously hidden PLANTS and FEEDSTOCK fields. There are also shortcuts for unhiding hidden fields referenced from the formula, or the hidden filter, indicated by the dashed cell icons (⬚).

## 3.4 Architecture

Our visual query system allows a large class of queries to be expressed by end users. As a necessary consequence, we can make few assumptions about how fast results can be computed. In many cases, even though the final query desired by the user may be cheap to compute, intermediate or explorative queries generated during interactive query building may be expensive. Intermediate queries may even contain user errors, such as circular dependencies in formulas. A key requirement of our system is to avoid getting the user stuck in such states, and to keep the query building interface responsive and up to date even when expensive or incorrect queries are encountered.

Our system's basic architectural decision is to defer all query processing to a relational database backend, generating SQL queries over JDBC and retrieving a complete new result every time the user modifies the query model. This produces transactionally consistent results while avoiding complicated incremental evaluation logic. We then provide the necessary smoke and mirrors to give the user an experience of responsive, incremental query building. The key features to this effect are as follows:

**Visual stability.** Our query semantics ensure that nested result tuples from successive steps of a visual query building process remain in the same order by default, and that the set of logical tuples in a relation does not usually change as fields are hidden or shown. The presentation properties of result layouts, such as table column widths, are based on average and confidence interval values that do not change once a target number of unique sample values have been collected from observed query results. Text breaking and font sizing is done to ensure that even exceptionally long string values can be displayed at a given visual width. Thus, even though an entirely new result set is generated every time the user modifies the query, the visual transition from the old to the new result appears seamless. The generation of compact nested table layouts is done using our previously described system for visualization of structured hierarchical reports [5].

**Decoupled query and result updates.** The display of a nested table header, which our system uses to communicate query state, need not be postponed until a query returns with actual results. Better yet, upon a change to the query model, we can immediately render a new table layout whose structure and indications are based on the updated query model, but whose data is taken from whichever query completed most recently. For fields not present in the old result, we show a placeholder icon (⋯) where data values would normally appear. Meanwhile, updated SQL queries run while a non-modal progress indication is shown in the toolbar area. Once the query completes, the result layout is rendered again with actual results. The user does not need to wait for the query to complete before making new changes.

For example: When unhiding a previously hidden field, the user sees the result layout immediately update to accommodate the new table column, already at the correct width, with placeholder icons seamlessly being replaced by data values as soon as the updated query completes.

**Interruptable queries.** If the user modifies the query before the previous query has finished executing, the previous query is automatically interrupted using the database backend's preferred mechanism. This is crucial for letting the user escape from long-running queries, and also allows the user to perform multiple modifications

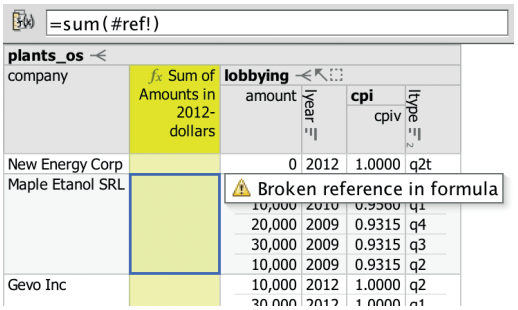



**Figure 4: High-level error handling. A referenced field was deleted, so the formula can no longer be evaluated. The system shows a warning while evaluating the rest of the query normally.**

to the query without waiting for the exact result of each step to appear. The on-screen layout remains undisturbed by the automatic interruption and restarting of queries in the background. Note that even long-running queries can be constructed with responsive result feedback if the user can manage to temporarily filter the dataset down to a smaller size during query construction.

**Automatic query limiting.** All generated SQL queries include an automatic LIMIT clause, retrieving initially 100 tuples total for each relation field. This populates the visible part of a typical result window. If the user scrolls far enough down to see the end of the result layout, and there are more tuples left, the query is re-executed using a limit twice as large as before. This allows the user to reach tuple $N$ in $O(N)$ time. Infinite scrolling appears seamless.

**High-level error handling.** User-defined formulas introduce a variety of possible error conditions, including circular references, broken references, type errors, and arithmetic runtime errors. Our system detects and handles many such errors at a high level. In the result layout, formulas with errors are highlighted in yellow, with a tooltip showing specific error messages if the cursor is moved to the highlighted area. For query evaluation purposes, erroneous formulas are replaced by null values, ensuring that the rest of the query can still be evaluated normally. See Figure 4.

Complete high-level error handling requires the set of functions and data types available in formulas to be known to the system. It may also require functions such as arithmetic division to be rewritten to return null instead of triggering runtime errors on, say, division by zero. SIEUFERD includes a standardized set of formula functions that can be compiled to the dialects of various database backends, currently PostgreSQL, MySQL, and Oracle. Standardizing functions and data types allows a single unit test suite and online documentation set to be used for all backend dialects.

**Undo/redo.** Undo/redo can be easily supported by storing successive states of the modified query model; a similar technique is used in Tableau [55, p. 90]. Like other kinds of query modifications, undo/redo benefits from several of the previously mentioned features, e.g. interruptable queries.

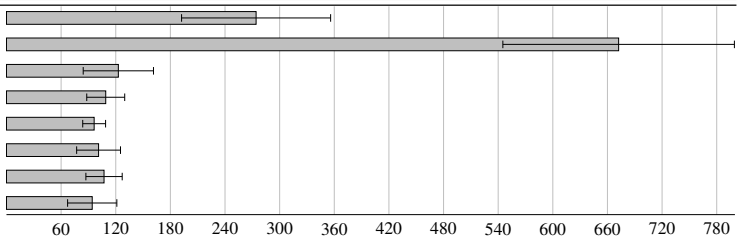
# 4. FORMATIVE USER STUDY

We conducted a formative user study with 14 participants (5 male, median age 42) from a variety of technical and professional backgrounds; see Table 5 (appendix) for a demographic summary.

In the first part of the study, done by users A-I, users were given standardized tasks aimed at assessing the initial learnability of our tool. No prior training was given; instead, initial tasks were designed to act as training tasks for subsequent ones. In the sec-

**Table 3: Tasks and timings for standardized tasks used as part of the formative user study. Error bars show the standard error of the mean.**

| # | Task | | Training for task | N | Mean time to complete task (s) |
|---|---|---|---|---|---|
| 1a | Lobbying Totals | Manual join | 2a | 7 | 274 |
| 1b | | Formula | 2b | 7 | 672 |
| 2a | Inflation Correction | Manual join | | 7 | 123 |
| 2b | | Formula | | 7 | 109 |
| 3 | Single-level auto join | | 4-6 | 7 | 96 |
| 4 | Multi-level auto join | | 5-6 | 7 | 101 |
| 5 | Filter via auto join | | | 7 | 107 |
| 6 | Multi-level auto join, again | | | 6 | 94 |




**Tasks done:** Tasks 1-2 were done by users A-G. Tasks 3-6 were given to users B-I, with some exceptions. **Task order:** User F did tasks 1-2 last. Order is otherwise as indicated. **Hints:** Training tasks included hints as necessary. In task 2a, users D and G were told that they would need to use the JOIN feature.

ond part of the study, and as time permitted during earlier sessions, users were given a chance to do more open-ended tasks on datasets we provided, including some datasets from the users' own organization. Here, we gave participants demos and instructions for operating our tool, in order to gather higher-level observations than would be possible during pure learning tasks.

From screen and voice recordings of each user study session, we collected detailed observations that were later coded and categorized, as well as timing data for standardized tasks.

## 4.1 Standardized Tasks

This section describes tasks and timings for the standardized portion of our study. We designed the standardized tasks to assess the initial learnability of our system's basic query operators, likely to cover a range of common queries. Tasks designated as training tasks reflect the user's first encounter with a particular feature, with few upfront instructions given on how to proceed. If a user got stuck during a training task, hints were given and any relevant observations noted, ensuring that the user progressed to the corresponding follow-up task. See Table 3.

**Formulas and manual joins.** Tasks 1 and 2 correspond to the lobbying example from Section 3.3, in two parts. In task 1, which functions as a training task, the user is started off with a fresh query showing only the PLANTS_OS table, and is asked to find the total amount spent on lobbying by each organization. A minimal schema diagram is provided on paper, showing the two tables involved and the fields to be matched in the join condition. The user will have to discover that the operation called JOIN is needed, and then figure out how to use a formula to calculate totals. In task 2, the user is asked to modify the existing query to calculate inflation-corrected totals, using consumer price index values from the CPI table and features that have already been used. The user now has to realize that another join is needed, followed by one or two additional formulas. This task tests whether the user, after only a single training task, has developed enough of a mental model of how joins and formulas work to combine the two features to arrive at a single result.

Task 1, the training task, took users about 16 minutes on average, with 70% of the time spent after users figured out the initial manual join. Task 2, a strictly harder task using the same features as task 1, took only about 4 minutes, 4.1 times faster. The difference is statistically significant ($p = 0.009$ with two-tailed Welch's t-test). Comparing only times spent on the join portion of the tasks vs. times spent on the formula portion of the tasks, the difference is only statistically significant for the latter ($p = 0.004$). In this case, users solved the second formula task 6.2 times faster than the first.

**Auto joins and filters.** Tasks 3-6 involve automatic joins over known foreign key relationships (*auto joins*), starting again from a fresh query showing a single base table. Users are given a schema diagram on paper, with the relevant joins marked, and told that because the system already knows about the relationships between the tables, it will not be necessary to use the manual JOIN action. Tasks 3 and 4 ask the user to produce a report-style query similar to the course catalog shown in Figure 2, first adding a table related to the base table via a single join (e.g. READINGS), and then adding a table related to the base table via multiple joins (SECTIONS, INSTRUCTORS_SECTIONS, INSTRUCTORS). In task 5, the user is asked to filter the result on a field in a table that has not yet been joined into the current query, specifically to "show only courses offered in Spring 06-07", where semester names are stored in a separate table. This allows us to assess the user's expectations wrt. interactions between nested joins and filters. In task 6, the user is started off with a fresh new query, starting from a different base table (INSTRUCTORS). Having previously produced a course catalog showing a list of instructors for each course, the user is now asked to show a list of courses taught by each instructor. This repeats task 4, but from the opposite end of the schema.

## 4.2 Observations

We now discuss a specific observations gathered from both the standardized and the open-ended portions of the study.

**Manual joins.** The manual join dialog, quoting user C, was "actually very easy to use"; most users moved through it quickly and correctly on their first attempt. Still, users preferred auto joins once introduced to them, see below. Users CEJ wanted to visually verify that the equijoin condition was satisfied, and were briefly confused because our system automatically hid the redundant constrained field on the nested side of the join. Users performing task 2 had no problems with the join portion of the task; only users DG required a hint that they would need to use the JOIN feature again, while the rest realized this on their own.

**Formulas.** When first attempting to perform a sum aggregation, users BCDE started by looking for an explicit sum action, as would be found in Excel's toolbar. Users CGK looked for an Excel-style formula builder. Having eventually realized that they needed to insert a calculated field and enter a formula themselves, users DEFK had initial trouble learning how to physically enter the formula, trying for example to enter the formula in an already-existing column, or in the column header.

In Excel, sums can be produced either using formulas or pivot tables. The two interfaces are largely separate, with users often preferring one or the other. Our system follows the formula approach. Users CH commented that they thought of pivot tables when first trying to compute a sum, while users BEI thought of pivot tables during other tasks.

A significant difference between spreadsheet formulas and SIEUFERD formulas is that the latter, like SQL queries, reference entire columns of values rather than an explicit range of cells. Users ABCFH expected this on their first attempts to insert a reference in a sum formula. Users DEGN expected the spreadsheet model, initially attempting to select a range of cells. A related challenge was to understand the level at which a calculated field should be inserted in order for sums to be grouped in the right way. The fact that the position of a formula in the relation hierarchy determines the grouping of aggregate functions is a further deviation from the spreadsheet model, while the lack of an explicit `GROUP BY` clause may be confusing to SQL users. User H tried to specify the set of columns to group by in the aggregate function itself, as in the formula =SUM([NAME],[AMOUNT]), while user F tried to hide every field other than the one to be summed. User G attempted to invoke the HIDE DUPLICATE ROWS action. Users CFGH also tried placing the calculated field next to the value to be summed rather than at the parent level. The latter has the trivial effect of producing sums each over only a single input value. User G, who spent 20 minutes on Task 2b, thought aloud:

*"Wouldn't it be fantastic if there was a way simply to operate at that group level rather than these individual entries? [After creating a new formula at the correct level:] Is it doing it that way? Oh, that's perfect. ... That is meeting my heart's desire. But I wouldn't have the cue for that."*

Despite initial difficulty with formulas in training task 1b, users applied them quickly and accurately in follow-up task 2b. This is despite the follow-up task requiring more steps (a join, a scalar function, and an aggregate function). This suggests users are able to apply formulas effectively after first learning them, but that there is significant potential for improved learnability. We agree with users AM, who suggested adding an explicit sum action like that of Excel. This feature would automatically generate a sum formula above the nearest one-to-many relationship, which would then serve as an example to the user to learn from.

After initial learning, users appreciated the behavior of formulas. Users CEGK noted explicitly that the behavior of aggregate functions, including grouping and subtotaling behavior, made sense. Users ILK also commented that the all-column nature of formula references made sense and was an advantage over Excel's range-style references. User K noted:

*"I just feel like I have a truer sense of what I'm adding up, or what's being considered in this format vs. the traditional Excel. Because [in Excel] you could be pulling from the wrong places, you can be getting weird numbers, you could accidentally hit a field that now ends up in your calculation."*

**Field selection; auto joins.** Users performing tasks 3-6, or similar tasks on other datasets, were generally able to use the auto join feature without trouble. The exception was user N, who had a hard time because of the lack of visible indications in the result area that more fields could be shown. User G also noted this issue. Users IKN specifically looked for an action named "Unhide", like in Excel. This suggests that our user interface needs a more visible affordance for accessing hidden fields. We expect hidden fields to be far more common in SIEUFERD than in Excel, since a typical database query projects only a small subset of columns available from instantiated database tables. The design of an improved unhide affordance should take this into account.

Users EGHJKL reacted particularly enthusiastically to the auto join feature, using words such as "fantastic", "wow", "damn", and "amazing". User E noted:

*"Yes, the manual join made sense, but that was a very simple situation. I wouldn't want to have done the joins on this [more complicated database]. The fact that I was just able to double-click and expand it out, that meant, it dumbed the task down to the level that I was happy performing it."*

**Field selection; efficiency.** One important problem was that of poor defaults for which fields should be visible immediately after a new relation is introduced into the current query. For manual joins, all (non-redundant) fields in the foreign table would be visible in the nested relation; this made it hard to grasp the overall structure of the query without first going through the step of hiding a number of irrelevant columns, usually necessitating horizontal scrolling. For auto joins, in contrast, only primary key fields were displayed by default. This also turned out to be a poor choice, because primary key fields often consist of purely technical identifiers that neither help the user identify an entity in the database nor its type. An example would be the relation EMPLOYEES(ID, FIRST_NAME, LAST_NAME), where the database identifies each tuple by the technical primary key ID (maybe a number, like "16") but where the user would rather like to see the first and last names of each employee—despite the theoretical possibility that two employees might have the same name. Showing only primary key fields by default made auto joins harder to work with than necessary, requiring users to click four or five times in the field selector in order to introduce a new relation and show a reasonable set of fields from that relation.

Post-study, in response to the problem of poor field visibility defaults, we modified our system to allow a subset of columns from each database table to be marked as human-readable heading fields. These are the fields that will initially be visible whenever the table in question is introduced into a query. As suggested by users MN, various heuristics can be used to configure this setting automatically. Several proposed attribute ranking algorithms [19, 44] could be suitable. For now, we simply look for column names containing the words "title" or "name". Summarization may even be useful in the vertical direction, with several techniques available [14, 51].

For databases containing a large number of fields per table, navigating the field selector became cumbersome. This was noted by users EJM, who got a chance to try our tool on a real data warehouse schema containing 22 interconnected tables with up to 40-73 fields each (19 on average). User L also pointed this out for the smaller course catalog schema. User E explains:

*"You've got massive lists, and they're not ordered alphabetically. You've got table names, and field names, and sometimes they are not very English."*

One part of the problem is that users spent a significant amount of time scanning up and down looking for specific field names. A search box in the field selector, like that of the filter popup, would help mitigate this, as suggested by users JM. A separate problem is the fact that the multitude of primitive fields in the field selector obscures the overall structure of relation fields in the query, including those accessible via auto joins. Users JM also commented that they would have liked to see a schema diagram of some sort on-screen. In the future, we may consider adding a second kind of field selector that shows relation fields only in a tree representation that is fully expanded by default; this would provide a compact way to see the entire foreign key structure of the database schema as reachable from the current query.

## 4.3 General Sentiment

At the end of the session, users CDHIJK expressed that they had a high degree of understanding of the tool. User K, who had 2-300 hours of experience with SQL from their previous job, noted:

*"It's probably fair to say that I am as comfortable with this as I am with SQL right now, just because I haven't used SQL that often*

*in the recent past. Given 2 hours, I think I could make an accurate report in this, allowing for mistakes, and fixing my mistakes. Take that same period in SQL, and I think I would still be at sea."*

Users EJKL rated SIEUFERD favorably compared to existing commercial tools they are familiar with.

User J: *"It took me a lot longer to get anything useful out of Access after I first started using that. So that's huge. This is more intuitive than either Excel or Access. I think, for the novice that doesn't know what they're doing, this can be very powerful."*

# 5. CONTROLLED USER STUDY

In a second user study, we aimed to get a more precise idea of how users might rate our system compared to an existing industry tool. We chose the "Query Design" facility of Microsoft Access 2016 as a control. Being part of the Office Professional suite, it is one of the most common visual query tools available. It is also a good example of a query builder that uses a diagram-based approach rather than direct manipulation of results (see Related Work).

The controlled study was a within-subjects counterbalanced design, measuring usability using the System Usability Scale (SUS) [8]. Tullis and Stetson [56] recommend sample sizes of 12-14 users to get reasonably representative results from within-subjects studies based on the SUS survey; we collected data from 14 users (5 male, median age 36). See Table 5 (appendix) for a demographic summary. Only users OTÆ had prior experience with the Access query designer. We met with each user for a single study session, structured as follows:

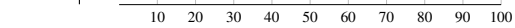
1. Complete demographic/background survey.
2. Briefly discuss the sample database that will be used for tasks, consulting a schema diagram on paper. The paper diagram remains available to the user during the tasks that follow.
3. Work through some standardized tasks to evaluate Tool 1. Stop after about 20 minutes. The first tool is SIEUFERD for half of the users and Microsoft Access for the other half, randomized.
4. Complete SUS survey for Tool 1.
5. Work through the same tasks in Tool 2, under otherwise identical conditions. Stop after about 20 minutes.
6. Complete SUS survey for Tool 2.
7. Discussion and feedback.

The standardized tasks, all done on the 7-table "Northwind" example database that shipped with older versions of Microsoft Access, are intended to be realistic examples of queries that a user might want to run on such a database. They incorporate joins, filters, sorting, scalar calculations and aggregates, but are limited to queries that can be expressed in Microsoft Access' visual query designer; this excludes queries requiring nested results as well as multi-block queries (e.g. aggregates used as inputs to other aggregates). The exact tasks are listed in Table 6 (appendix). In both tools, we configured foreign key relationships upfront so that the user would not have to manually specify exact join constraints between tables. The first five tasks are guided training tasks, intended to expose the user to all features, in both tools, that are needed to complete the subsequent unguided tasks. The guided tasks tended to take about half of the 20 minutes that users had available to try each tool. After the guided tasks, users were asked to try solving four unguided tasks without help. Since the main purpose of tasks was to give the user enough of an impression of each system to complete the subsequent SUS survey, we gave hints during unguided tasks whenever users reported being stuck.

The results of the study are shown in Table 4. The raw SUS score is reported along with separate Learnability and Usability scores as defined by Lewis and Sauro [39], as well as a percentile rating among 30 other studies in the B2B (Business Software) category as detailed by Sauro [49]. The difference in raw SUS scores between Access and SIEUFERD is statistically significant ($p = 0.0019$ with two-tailed paired t-test).

**Table 4: Mean SUS survey results for the controlled study, using various standard scales. Higher scores are better. Error bars show the standard error of the mean.**

| *Scale* | *Tool* | *Score (0-100)* |
|---|---|---|
| Raw SUS | Access | 50 |
| | Sieuferd | 68 |
| Learnability | Access | 49 |
| | Sieuferd | 64 |
| Usability | Access | 50 |
| | Sieuferd | 69 |
| Percentile | Access | 6 |
| | Sieuferd | 52 |




Interpreting the results, with the caveat that these observations are based on only 20-minute interactions with each tool, we see that SIEUFERD significantly outperformed Microsoft Access in terms of usability. Most of the difference can be attributed to the poor performance of Microsoft Access, considering its low ranking on the percentile scale; SIEUFERD simply achieved an average rating compared to other business software. This supports the original hypothesis of our paper: database querying is hard, but can be made significantly easier using a direct manipulation interface. SIEUFERD still has significant potential for improved usability. In conversations with users, the main requests for future design improvements were (1) the ability to get an overview of the complete database schema from within the query interface and (2) reduced dependency on formulas during query building. This is consistent with observations from the formative study.

# 6. CONCLUSION

SIEUFERD is a visual query system that achieves SQL-like expressiveness from a pure direct manipulation interface. Whereas previous direct manipulation systems either sacrifice expressiveness or hide the actual query from the user, SIEUFERD integrates the query and its result into a single interactive visualization, using spreadsheet concepts like filters and formulas to expose the complete state of the current query. Compared with the diagram-based query designer of Microsoft Access 2016, users greatly preferred our direct manipulation interface, with the latter scoring 46 percentiles higher on a SUS-based percentile scale. For data-minded people of all professions, we believe SIEUFERD's interaction style holds promise as an alternative to hand-coded SQL.

# 7. FUTURE WORK

In the current query interface, some queries are expressible yet awkward to construct; examples include range filters, grouping on custom attributes, and `UNION`-type queries. Here, the interface could be improved without significant changes to the underlying query model. See for example the proposed syntactic sugar for union queries in Figure 7 (appendix). Other future work will focus on expanding the ways in which results can be displayed; our query model is well-suited for the retrieval of data for visualizations such as dashboards, crosstabs, calendars, forms, and reports. Already supported are crosstabs, see Figure 5 (appendix), and form/report layouts [5]. Finally, we hope to incorporate *editing* of data; this will allow SIEUFERD to act as a complete schema-independent end user front-end for relational databases.

# 8. REFERENCES


[1] A. Abouzied, J. 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 '12)*, pages 207–218, New York, NY, USA, 2012. ACM.

[2] S. Achler. GBXT: A gesture-based data exploration tool for your favorite database system. In *Model and Data Engineering*, pages 224–237. Springer International Publishing, Cham, Switzerland, 2014.

[3] M. Angelaccio, T. Catarci, and G. Santucci. Query by Diagram: A fully visual query system. *Journal of Visual Languages & Computing*, 1(3):255–273, 1990.

[4] E. Bakke and E. Benson. The schema-independent database UI: A proposed holy grail and some suggestions. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, 2011.

[5] E. Bakke, D. R. Karger, and R. C. Miller. Automatic layout of structured hierarchical reports. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2586–2595, December 2013.

[6] E. Baralis and J. Widom. An algebraic approach to static analysis of active database rules. *ACM Transactions on Database Systems (TODS)*, 25(3):269–332, September 2000.

[7] T. Berners-Lee, Y. Chen, L. Chilton, D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, and D. Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *Proceedings of the 3rd International Semantic Web User Interaction Workshop (SWUI '06)*, 2006.

[8] J. Brooke. SUS: A quick and dirty usability scale. In *Usability evaluation in industry*, pages 189–194. Tailor & Francis, London, UK, 1996.

[9] M. Burnett, J. Atwood, R. Walpole Djang, J. Reichwein, H. Gottfried, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11:155–206, March 2001.

[10] B. Cao and A. Badia. SQL query optimization through nested relational algebra. *ACM Transactions on Database Systems (TODS)*, 32(3):18, 2007.

[11] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages & Computing*, 8(2):215–260, 1997.

[12] J. P. Caulkins, E. L. Morrison, and T. Weidemann. Spreadsheet errors and decision making: Evidence from field interviews. *Journal of Organizational and End User Computing*, 19(3):1, 2007.

[13] K. S.-P. Chang and B. A. Myers. Using and exploring hierarchical data in spreadsheets. In *Proceedings of the 34th Annual ACM Conference on Human Factors in Computing Systems (CHI '16)*, New York, NY, USA, 2016. ACM.

[14] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)*, pages 888–899. VLDB Endowment, 2004.

[15] W.-K. Chen and P.-Y. Tu. VisualTPL: A visual dataflow language for report data transformation. *Journal of Visual Languages & Computing*, 25(3):210–226, 2014.

[16] C. Clack and L. Braine. Object-oriented functional spreadsheets. In *Proceedings of the 10th Glasgow Workshop on Functional Programming (GlaFP '97)*, 1997.

[17] E. F. Codd. Relational completeness of data base sublanguages. In *Database Systems*, pages 65–98. Prentice Hall, 1972.

[18] R. Collie. The 3rd most common button in data apps is... http://www.powerpivotpro.com/2012/03/the-3rd-most-common-button-in-data-apps-is, March 2012.

[19] G. Das, V. Hristidis, N. Kapoor, and S. Sudarshan. Ordering the attributes of query results. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 395–406, New York, NY, USA, 2006. ACM.

[20] E. Díaz-Struck. Ethanol industry battles to keep incentives, May 2013. Investigation for the New England Center for Investigative Reporting and Connectas, available at http://eye.necir.org/2013/05/26/ethanol-industry-battles-to-keep-incentives.

[21] S. El-Mahgary and E. Soisalon-Soininen. A form-based query interface for complex queries. *Journal of Visual Languages & Computing*, 29:15–53, 2015.

[22] R. G. Epstein. The TableTalk query language. *Journal of Visual Languages & Computing*, 2(2):115–141, 1991.

[23] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems (TODS)*, 2(3):262–278, 1977.

[24] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2009.

[25] Y. Han, G. Wang, G. Ji, and P. Zhang. Situational data integration with data services and nested table. *Service Oriented Computing and Applications*, 7(2):129–150, 2013.

[26] L. Hella, L. Libkin, J. Nurmonen, and L. Wong. Logics with aggregate operators. *Journal of the ACM (JACM)*, 48(4):880–907, July 2001.

[27] G.-J. Houben and J. Paredaens. A graphical interface formalism: Specifying nested relational databases. In *Proceedings of the IFIP TC2 Working Conference on Visual Database Systems*, pages 257–276, 1989.

[28] Y. E. Ioannidis. Visual user interfaces for database systems. *ACM Computing Surveys (CSUR)*, 28(4es), 1996.

[29] G. Jaeschke and H. J. Schek. Remarks on the algebra of non first normal form relations. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '82)*, pages 124–138, New York, NY, USA, 1982. ACM.

[30] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 13–24, New York, NY, USA, 2007. ACM.

[31] D. Janvrin and J. Morrison. Using a structured design approach to reduce risks in end user spreadsheet development. *Information & management*, 37(1):1–12, 2000.

[32] M. Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. *Proceedings of the VLDB Endowment*, 1:695–709, August 2008.

[33] M. Jayapandian and H. V. Jagadish. Expressive query specification through form customization. In *Proceedings of the 11th International Conference on Extending Database Technology (EDBT '08)*, pages 416–427, New York, NY, USA, 2008. ACM.

[34] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the 2011 annual conference on Human Factors in Computing Systems (CHI '11)*, pages 3363–3372, New York, NY, USA, 2011. ACM.

[35] E. Kandogan, E. Haber, R. Barrett, A. Cypher, P. Maglio, and H. Zhao. A1: End-user programming for web-based system administration. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology (UIST '05)*, pages 211–220, New York, NY, USA, 2005. ACM.

[36] K. Kowalzcykowski, A. Deutsch, K. W. Ong, Y. Papakonstantinou, K. K. Zhao, and M. Petropoulos. Do-It-Yourself database-driven web applications. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR '09)*, 2009.

[37] D. Król, J. Oleksy, M. Podyma, and B. Trawiński. The analysis of reporting tools for a cadastre information system. In *Proceedings of the 9th International Conference on Business Information Systems (BIS '06)*, pages 150–163, 2006.

[38] M. Levene. *The Nested Universal Relation Database Model*, volume 595 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg, 1992.

[39] J. R. Lewis and J. Sauro. The factor structure of the system usability scale. In *Proceedings of the 1st International Conference on Human Centered Design (HCD '09)/HCI International 2009*, pages 94–103, Berlin, Heidelberg, 2009. Springer-Verlag.

[40] L. Libkin and L. Wong. On the power of aggregation in relational query languages. In S. Cluet and R. Hull, editors, *Proceedings of the 6th International Workshop on Database Programming Languages*

*(DBPL '97)*, Lecture Notes in Computer Science, pages 260–280. Springer Berlin/Heidelberg, 1998.

[41] B. Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *Proceedings of the IEEE 25th International Conference on Data Engineering (ICDE '09)*, pages 417–428, April 2009.

[42] N. Lorentzos and K. Dondis. Query by Example for Nested Tables. In *Database and Expert Systems Applications*, pages 716–725. Springer, 1998.

[43] R. M. McCutchen, S. Itzhaky, and D. Jackson. Initial report on Object Spreadsheets. Technical Report MIT-CSAIL-TR-2016-001, MIT Computer Science and Artificial Intelligence Laboratory, January 2016.

[44] M. Miah, G. Das, V. Hristidis, and H. Mannila. Standing out in a crowd: Selecting attributes for maximum visibility. In *Proceedings of the 24th International Conference on Data Engineering (ICDE '08)*, pages 356–365, Washington, DC, USA, April 2008. IEEE Computer Society.

[45] A. Nandi, L. Jiang, and M. Mandel. Gestural query specification. *Proceedings of the VLDB Endowment*, 7(4):289–300, 2013.

[46] Y. Papakonstantinou, M. Petropoulos, and V. Vassalos. QURSED: Querying and reporting semistructured data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 192–203, New York, NY, USA, 2002. ACM.

[47] S. Polyviou, G. Samaras, and P. Evripidou. A relationally complete visual query language for heterogeneous data sources and pervasive querying. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 471–482, Washington, DC, USA, 2005. IEEE Computer Society.

[48] L. Qian, K. LeFevre, and H. V. Jagadish. CRIUS: User-friendly database design. *Proceedings of the VLDB Endowment*, 4(2):81–92, 2010.

[49] J. Sauro. *A practical guide to the System Usability Scale: Background, benchmarks & best practices*. Measuring Usability LLC, 2011.

[50] B. Shneiderman. Direct Manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[51] M. Singh, A. Nandi, and H. V. Jagadish. Skimmer: Rapid scrolling of relational query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 181–192, New York, NY, USA, 2012. ACM.

[52] M. Spenke and C. Beilken. A spreadsheet interface for logic programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '89)*, pages 75–80, New York, NY, USA, 1989. ACM.

[53] H. J. Steenhagen, P. M. G. Apers, and H. M. Blanken. Optimization of nested queries in a complex object model. In *Proceedings of the 4th International Conference on Extending Database Technology (EDBT '94)*, pages 337–350, New York, NY, USA, 1994. Springer New York.

[54] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional databases. *Communications of the ACM*, 51(11):75–84, November 2008.

[55] C. R. Stolte. *Query, analysis, and visualization of multidimensional databases*. PhD thesis, Stanford University, Stanford, CA, USA, 2003.

[56] T. S. Tullis and J. N. Stetson. A comparison of questionnaires for assessing website usability, 2004. Usability Professionals Association (UPA) 2004 Conference.

[57] J. Tyszkiewicz. Spreadsheet as a relational database engine. In *Proceedings of the 2010 International Conference on Management of Data (SIGMOD '10)*, pages 195–206, New York, NY, USA, 2010. ACM.

[58] J. Van den Bussche and S. Vansummeren. Translating SQL into the relational algebra. Course notes, Hasselt University and the Free University of Brussels, retrieved April 2016. http://cs.ulb.ac.be/public/_media/teaching/infoh417/sql2alg_eng.pdf.

[59] L. Wegner, S. Thelemann, J. Thamm, D. Wilke, and S. Wilke. Navigational exploration and declarative queries in a prototype for visual information systems. In C. Leung, editor, *Visual Information Systems*, volume 1306 of *Lecture Notes in Computer Science*, pages 199–218. Springer Berlin/Heidelberg, 1997.

[60] F. Yang, N. Gupta, C. Botev, E. F. Churchill, G. Levchenko, and J. Shanmugasundaram. WYSIWYG development of data driven web applications. *Proceedings of the VLDB Endowment*, 1(1):163–175, 2008.

[61] M. M. Zloof. Query-by-Example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

# APPENDIX

## A. EXPRESSIVENESS

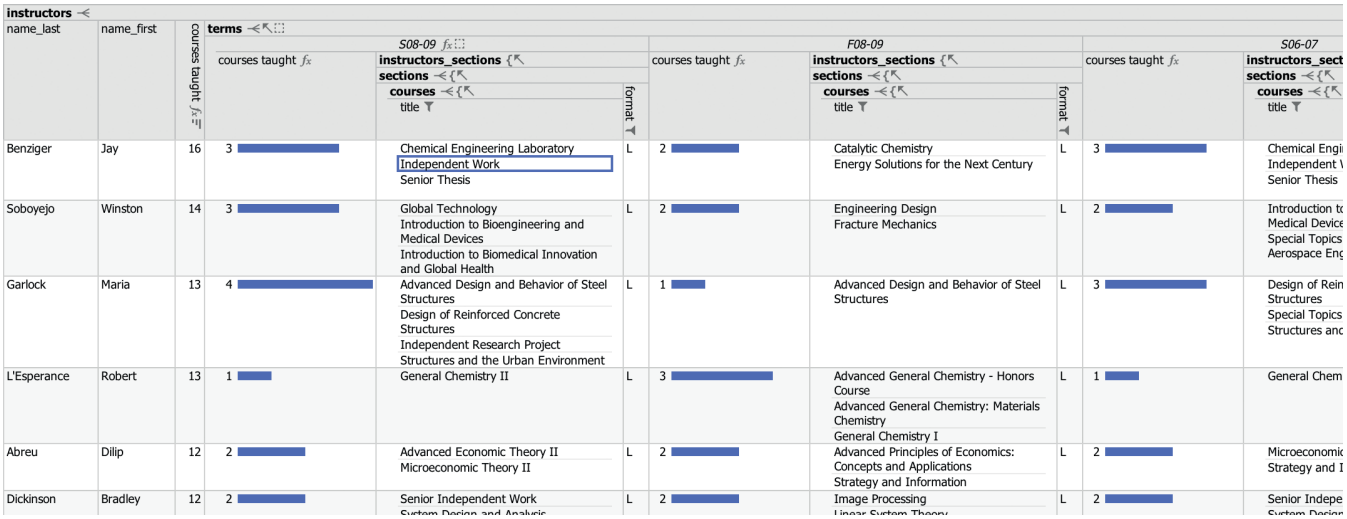
Like Liu and Jagadish [41], we demonstrate relational completeness of our visual query language by defining a translation from a *complete set* of operators in the relational algebra ($\sigma\pi\times\cup-$) to queries in our visual language. We also translate outer joins as well as the *extended projection* and *grouping* operators [24, p. 213]; the latter two formalize scalar and aggregate calculations, respectively. Assume set semantics in the relational algebra.

**Notation.** Let $e$, $e_a$, and $e_b$ be relational algebra expressions. Let $N(e)$ be the number of attributes in $e$. Assume that the attribute names of any relational algebra expression $e$ are $e[1], ..., e[N(e)]$. Define a *formula*, notated $\langle ... \rangle$, to be a functional expression over attribute names. Formulas are used both in the relational algebra and in the SIEUFERD query model. Properties in the query model are used as defined in Table 2.

**Translation from relational algebra.** Let $t(e)$ be a translation from a relational algebra expression $e$ to a relation field in the SIEUFERD query model. We define $t(e)$ recursively as follows:

- *Constants.* If $e = U$, where $U$ is a constant relation, then $t(e)$ is a relation field with INSTANTIATEDTABLE $= U$. It has primitive child fields named $e[1], ..., e[N(e)]$ with COLUMNDEFINITION set to the technical column names $U[1], ..., U[N(e)]$, respectively.
- *Selection.* If $e = \sigma_C(e_a)$, where $C$ is a boolean formula, then $t(e)$ is a relation field with the following child fields:
  - A relation field $t(e_a)$.
  - Primitive fields named $e[1], ..., e[N(e)]$ having COLUMNDEFINITION $= \langle e_a[1] \rangle, ..., \langle e_a[N(e)] \rangle$, respectively.
  - A primitive field with COLUMNDEFINITION $= C$, VISIBLE turned off, and FILTER set to include only values of TRUE.
- *Inner/outer joins, and Cartesian product.* If $e = e_a \bowtie_C e_b$, where $\bowtie$ is either an inner join or left outer join and $C$ is a boolean formula over attribute names in $e_a$ and $e_b$, then $t(e)$ is a relation field with the following child fields:
  - A relation field $t(e_a)$.
  - A relation field $t(\sigma_C(e_b))$ having HIDEPARENTIFEMPTY turned on iff $\bowtie$ is an inner join. The translation for $\sigma_C(e_b)$ applies even though $C$ may reference attributes outside of $e_b$.
  - Primitive fields named $e[1], ..., e[N(e)]$ having COLUMNDEFINITION $= \langle e_a[1] \rangle, ..., \langle e_a[N(e_a)] \rangle, \langle e_b[1] \rangle, ..., \langle e_b[N(e_b)] \rangle$, respectively.

  The Cartesian product ($\times$) is an inner join with $C = \langle$TRUE$\rangle$. A full outer join is the union of two left joins.



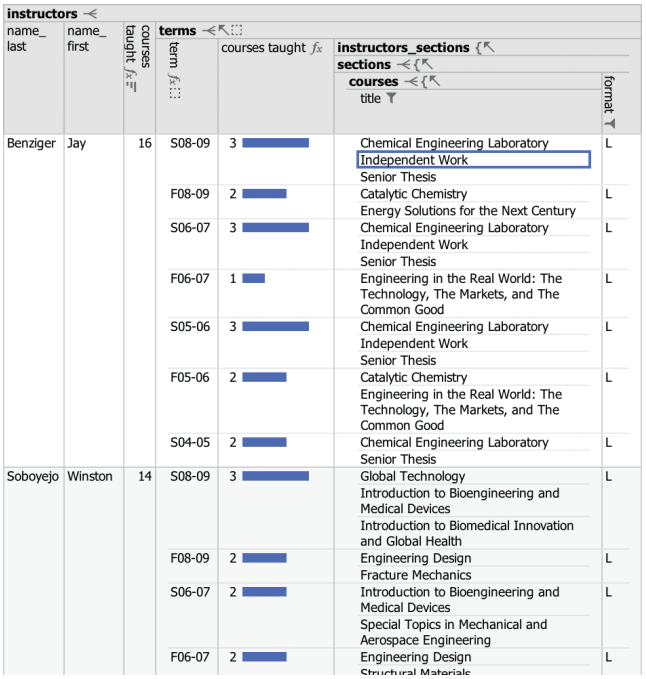
| instructors | | | terms | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | S08-09 | | | | F08-09 | | | | S06-07 | | |
| name_last | name_first | courses taught | courses taught | instructors_sections → sections → courses → title | format | courses taught | instructors_sections → sections → courses → title | format | courses taught | instructors_sect → sections → courses → title |
| Benziger | Jay | 16 | 3 | Chemical Engineering Laboratory<br>Independent Work<br>Senior Thesis | L | 2 | Catalytic Chemistry<br>Energy Solutions for the Next Century | L | 3 | Chemical Engi<br>Independent<br>Senior Thesis |
| Soboyejo | Winston | 14 | 3 | Global Technology<br>Introduction to Bioengineering and Medical Devices<br>Introduction to Biomedical Innovation and Global Health | L | 2 | Engineering Design<br>Fracture Mechanics | L | 2 | Introduction t<br>Medical Device<br>Special Topics<br>Aerospace Eng |
| Garlock | Maria | 13 | 4 | Advanced Design and Behavior of Steel Structures<br>Design of Reinforced Concrete Structures<br>Independent Research Project<br>Structures and the Urban Environment | L | 1 | Advanced Design and Behavior of Steel Structures | L | 3 | Design of Rein<br>Structures<br>Special Topics<br>Structures and |
| L'Esperance | Robert | 13 | 1 | General Chemistry II | L | 3 | Advanced General Chemistry - Honors Course<br>Advanced General Chemistry: Materials Chemistry<br>General Chemistry I | L | 1 | General Chem |
| Abreu | Dilip | 12 | 2 | Advanced Economic Theory II<br>Microeconomic Theory II | L | 2 | Advanced Principles of Economics: Concepts and Applications<br>Strategy and Information | L | 2 | Microeconomic<br>Strategy and I |
| Dickinson | Bradley | 12 | 2 | Senior Independent Work | L | 2 | Image Processing | L | 2 | Senior Indepe |

**Figure 5: A crosstab, shown as an example of one of the many alternative layouts that can be used to render data retrieved using the SIEUFERD query interface. The query shown here is identical to that of Figure 6, but has a crosstab formatting option enabled on the TERMS relation. All the usual query interface actions remain available from the crosstab layout.**

- *Extended projection.* If $e = \pi_{F_1 \to e[1], ..., F_n \to e[n]}(e_a)$ where each of $F_1, ..., F_n$ is a formula over attribute names in $e_a$, then $t(e)$ is a relation field with the following child fields:
  - A relation field $t(e_a)$.
  - Primitive fields named $e[1], ..., e[n]$, with COLUMNDEFINITION set to formulas $F_1, ..., F_n$, respectively.
- *Grouping (aggregation).* If $e = \gamma_{A_1, ..., A_n}(e_a)$, where each of $A_1, ..., A_n$ is either a grouping attribute name or an aggregation operator applied to an attribute name in $e_a$, then we can use the same translation as for extended projection by permitting aggregate functions in formulas. In this case, $t(e) = t(\pi_{\langle A_1 \rangle \to e[1], ..., \langle A_n \rangle \to e[n]}(e_a))$.
- *Set union.* A conditional formula can be used with a Cartesian product to produce the desired effect. If $e = e_a \cup e_b$, with $n = N(e)$, then $t(e) = t(\pi_{F_1 \to e[1], ..., F_n \to e[n]}(e_a \times e_b \times V))$ where $V$ is the constant relation $\{(\text{FALSE}), (\text{TRUE})\}$ and $F_i$ denotes the formula $\langle V[1] \; ? \; e_a[i] : e_b[i] \rangle$. In the future, we might introduce an explicit UNION function as syntactic sugar for this kind of construction; see Figure 7 for an example.
- *Set difference.* Here, we can filter for null values generated by a left join. If $e = e_a - e_b$, with $n = N(e)$, then $t(e) = t(\pi_{\langle e_a[1] \rangle \to e[1], ..., \langle e_a[n] \rangle \to e[n]}(\sigma_{\langle M \text{ IS NULL} \rangle}(e_a \bowtie_C e'_b)))$ where $\bowtie$ is a left outer join, $C = \langle e_a[1] = e'_b[1] \wedge ... \wedge e_a[n] = e'_b[n] \rangle$, and $e'_b$ adds an arbitrary non-nullable attribute $M$ to $e_b$, e.g. $e'_b = \pi_{\langle e_b[1] \rangle \to e'_b[1], ..., \langle e_b[n] \rangle \to e'_b[n], \langle 42 \rangle \to M}(e_b)$. Another approach would be to COUNT values in $e_b$ and filter for zero.

In the query model translations above, except when mentioned, the FILTER, SORT, JOINEDON, and INSTANTIATEDTABLE properties are cleared, while the VISIBLE, COLLAPSEDUPLICATEROWS, and HIDEPARENTIFEMPTY properties are TRUE.

Note that queries created by the fully general translation above can usually be simplified, e.g. by combining selection, projection, and table instantiation in a single relation field, or by using the JOINEDON property instead of filters on formula fields.



| instructors | | | terms | | | | |
|---|---|---|---|---|---|---|---|
| name_last | name_first | courses taught | term | courses taught | instructors_sections → sections → courses → title | format |
| Benziger | Jay | 16 | S08-09 | 3 | Chemical Engineering Laboratory<br>Independent Work<br>Senior Thesis | L |
| | | | F08-09 | 2 | Catalytic Chemistry<br>Energy Solutions for the Next Century | L |
| | | | S06-07 | 3 | Chemical Engineering Laboratory<br>Independent Work<br>Senior Thesis | L |
| | | | F06-07 | 1 | Engineering in the Real World: The Technology, The Markets, and The Common Good | L |
| | | | S05-06 | 3 | Chemical Engineering Laboratory<br>Independent Work<br>Senior Thesis | L |
| | | | F05-06 | 2 | Catalytic Chemistry<br>Engineering in the Real World: The Technology, The Markets, and The Common Good | L |
| | | | S04-05 | 2 | Chemical Engineering Laboratory<br>Senior Thesis | L |
| Soboyejo | Winston | 14 | S08-09 | 3 | Global Technology<br>Introduction to Bioengineering and Medical Devices<br>Introduction to Biomedical Innovation and Global Health | L |
| | | | F08-09 | 2 | Engineering Design<br>Fracture Mechanics | L |
| | | | S06-07 | 2 | Introduction to Bioengineering and Medical Devices<br>Special Topics in Mechanical and Aerospace Engineering | L |
| | | | F06-07 | 2 | Engineering Design<br>Structural Materials | L |

**Figure 6: Input data for the crosstab example in Figure 5, shown here in a regular nested table layout. The field COURSES TAUGHT has a formatting option enabled on it to display numbers using a bar chart visualization.**

**Table 5: User study participants and backgrounds. Users A-N participated in the formative study, users O-Ø in the controlled study.**

| # | *Professional Area* | *Educational Background* | *Technical Background/Tools Used* | | | |
|---|---|---|---|---|---|---|
| | | | *Excel* | *SQL* | *Programming* | *Other Tools* |
| A | Data journalism | Journalism | Daily | Weekly | A bit of Python | Access often, Tableau/OpenRefine occasionally |
| B | Business intelligence | Linguistics | Daily | Daily | Some PHP | BrioQuery daily |
| C | Business intelligence | Psychology | Daily | No | No | Spotfire daily, BrioQuery occasionally |
| D | Financial | Philosophy, Research Adm. | Daily | No | Learning basic Python | Some SAPGUI |
| E | IT decision-making | Computer Science | Weekly | In 1992 | Java/VB years ago | Some R |
| F | Business intelligence | Operations Mgmt., Business | Daily | Frequently | VB.net | BrioQuery daily |
| G | CS research, teaching | CS, Commun., Art Hist., Lit. | Monthly | Monthly | Java/C years ago | R a long time ago |
| H | Business intelligence | Sociology, Higher Education | Daily | No | No | BrioQuery daily |
| I | Health policy | Sociology | Weekly | No | No | Access for survey entry once, SPSS in school |
| J | Investigative journalism | English, Business/Econ. Journ. | Daily | No | Very basic Python | Access weekly, e.g. for joins before continuing in Excel |
| K | Publishing | Writing, Literature & Publishing | Daily | Frequently* | No | Crystal Rep. frequently*, 2 industry-specific systems |
| L | Health policy, research | Public Health, Public Policy | Daily* | No | No | Access for data entry*, knows SAS/Stata/SPSS/ArcGIS |
| M | Engineering data analytics | Finance, Management | Daily | Frequently* | Python/VB years ago | Access/Crystal Rep. frequently*; now Tableau, Alteryx |
| N | University administration | Math & Economics, Higher Ed. | Daily | No | No | BrioQuery for canned reports, internal CRUD apps |
| O | IT | Computer Science | Daily | Yearly | Java/VB/Perl* | Access monthly* |
| P | Bioinformatics | CS, Bioinformatics | Monthly | Weekly | Weekly | R monthly |
| Q | Electrical eng./research | Electrical Eng., Systems Eng. | Weekly | Tried once | C/Java/Fortran* | MATLAB frequently* |
| R | Medicine | MD, Adm. & Management | Weekly | No | No | Electronic medical records |
| S | Bioinformatics/research | Bioinformatics | Weekly | Tried twice | Daily | R daily, MATLAB |
| T | Bioinformatics/research | Bioinformatics | Daily | Monthly | Daily (Python, JS) | Access monthly*, Spotfire daily*, R weekly |
| U | Biomedical/data science | Chemical Eng./Statistics | Monthly | Daily | Some Python | R weekly, Access for data entry* |
| V | Student | Neuroscience | Weekly | No | Some Python | Some MATLAB, SPSS, Access for data entry |
| W | Library adm./info science | Biology | Weekly | No | No | BrioQuery monthly*, SAP, FileMaker |
| X | Student | Electrical Engineering & CS | Monthly | Once | Daily | R once, MATLAB |
| Y | Student | Journalism | Monthly | One course | One course (Java) | N/A |
| Z | Student | Journalism | Weekly | No | No | N/A |
| Æ | Journalism, teaching | Journalism, Law | Weekly | Monthly | No | Access*/OpenRefine/Tableau monthly, many others |
| Ø | Research | Electrical Engineering | Weekly | No | Weekly (Python, C++) | MATLAB, Access for data entry*, R/SAS/SPSS |

*In previous job.

**Table 6: Tasks used in the controlled study. Some additional bonus tasks were also available to users who finished quickly.**

| *Type* | # | *Task* | *Operations involved* |
|---|---|---|---|
| Guided | 1 | Show a list of products with the PRODUCTNAME and DISCONTINUED fields visible. | Field selection |
| | 2 | Find the total quantity sold of each product, via the quantities in the ORDERDETAILS table. | Join, aggregate |
| | 3 | Find the total sales for each product. This involves a UNITPRICE $*$ QUANTITY calculation. | Scalar formula, aggregate |
| | 4 | Include in totals only orders shipped outside the US. | Join, pre-aggregate filter |
| | 5 | Show the products with the most revenue first, hiding any order details if still visible. | Sorting |
| Unguided | 6 | Show customers and all their orders, sorted by customer. | Field selection, join, sorting |
| | 7 | For each of the customers' orders, show the total dollar amount for that order. | Join, scalar formula, aggregate |
| | 8 | Show the name and phone number of the shipping company serving each order. | Join, field selection |
| | 9 | Show only orders assigned to employee Margaret Peacock. | Join, filter |

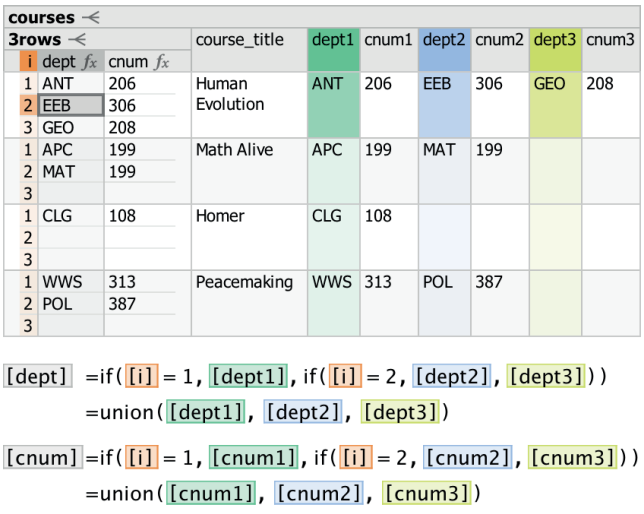



**Figure 7: A union query. Following a classic schema design antipattern, the COURSES table stores course codes using numbered table columns. To facilitate subsequent operations such as filtering by course code, the query collects course codes under a single nested relation via the helper table 3ROWS $= \{(1), (2), (3)\}$. An explicit UNION function, as proposed above, would make the expression of such queries more elegant.**