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.

This work was previously published at SIGMOD 2016.

1. INTRODUCTION

Four decades after Query by Example [51], the broad problem of Making Database Systems Usable [25] 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 [32, 27, 4]. Queries that involve “complex aggregates, nesting, correlation, and several other features remain on a tall pedestal approachable only by the initiated” [23]. 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... after OK and Cancel”¹. Spreadsheets lack basic database functionality such as joins and views, but demonstrate the great value of usable, general-purpose data manipulation tools [4].

Shneiderman [43] 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 [35] 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 [35]. 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. [35]

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-find), the first visual query system to meet all of the requirements above in

¹http://www.powerpivotpro.com/2012/03/the-3rd-most-common-button-in-data-apps-is

Short version prepared for the 2016 KDD IDEA Workshop.  
To cite, please refer to our full SIGMOD paper [5].  
KDD 2016 Workshop on Interactive Data Exploration and Analytics (IDEA'16). August 14th, 2016, San Francisco, CA, USA.
### Table 1: Summary of related systems, evaluated as visual query interfaces.

<table>
<thead>
<tr>
<th>Direct Manip.</th>
<th>Query Representation</th>
<th>Year</th>
<th>System</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>Unrestricted Nesting</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>on Result</td>
<td>2012</td>
<td>DataPlay</td>
<td>[1]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>Tabulator</td>
<td>[8]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spreadsheet</td>
<td>2016</td>
<td>Object Spreadsheets</td>
<td>[37]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Formulas</td>
<td>2010</td>
<td>Spreads as DB</td>
<td>[48]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1994</td>
<td>Forms3</td>
<td>[10]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exposed</td>
<td>2013</td>
<td>Mushroom</td>
<td>[20]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algebraic</td>
<td>2011</td>
<td>Wrangler</td>
<td>[29]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1991</td>
<td>TableTalk</td>
<td>[18]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algebraic</td>
<td>2013</td>
<td>GestureDB</td>
<td>[38]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>CRUIS</td>
<td>[41]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>SheetMusiq</td>
<td>[35]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>AppForge</td>
<td>[50]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1989</td>
<td>R</td>
<td>[22]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>App2You</td>
<td>[31]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>QB</td>
<td>[40]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2002</td>
<td>QURSED</td>
<td>[39]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form-based</td>
<td>2008</td>
<td>FormCust.</td>
<td>[28]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>QBN</td>
<td>[56]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1997</td>
<td>ESCHER</td>
<td>[49]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1989</td>
<td>PERPLEX</td>
<td>[44]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1977</td>
<td>QBE</td>
<td>[51]</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 2. RELATED WORK

Visual query systems have been surveyed by Catarci et al. [11] and, recently, El-Mahgry and Soisalon-Soininen [17]. Systems discussed in this section include, in particular, those that employ direct manipulation, nested results, or optimizations for traversing relationships in the database. Table 1 categorizes systems by query representation style, and provides an assessment of each system against the requirements set forth in the introduction.

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” [6]. Systems that base their result representation on a single flat table of primitive values, such as Tableau [46], 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 [19] in the flattened result (PARTS$\rightarrow$SUPPLIERS and PARTS$\rightarrow$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$\rightarrow$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.

We first discuss visual query systems that do not fall in the direct manipulation category. Form-based systems originated with Query by Example (QBE) [51], where the user populates a set of empty skeleton tables with conditions, variables (examples), and output indications. ESCHER [49] and QBN [36] extend QBE to support nested results, while PERPLEX [44] 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 [28] 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) [40], QURSED [39], and App2You [31], or through a diagrammatic query plan, as in VisualTPL [14]. The diagram-based query building style is common in commercial tools—Microsoft Access, Navicat, pgAdmin, dbForge, Alleryx 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 ear-

---

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 [24, 33], and display results using a nested table layout [6]. 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 [16] set of query operators plus calculation, aggregation, outer joins, sorting, and nesting [5, Appendix A]. This covers the full set of query operators generally considered as the minimum to model SQL [7, 21], and expresses, for example, all SELECT statements valid in SQL-92.

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 SIEUERED and the query designer found in Microsoft Access on the System Usability Scale (SUS) [9] after doing a series of tasks on each. Users rated SIEUERED 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.

This work was previously published at SIGMOD 2016 [5].
lier without redoing the steps afterwards” [35] (requirement R2). For example, in GestureDB [38], 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 R2 [22], AppForge [50], CRIUS [41], and Gneiss [13]. SheetMusiq [35] 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, or set difference. In other systems, the underlying algebraic expression is exposed directly, as in the procedural data manipulation scripts of Wrangler [29], the XQuery-like mashup scripts of Mashroom [20], or the diagram-based representation in TableTalk [18]. 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 [48] 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 [10], Object Oriented Functional Spreadsheets [15], AI [30], and Object Spreadsheets [37].

Lastly, 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 that current such representations are not expressive enough to support arbitrary queries (requirement R3). For example, the direct manipulation interfaces of Tabulator [8] 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 [46] allows a large class of two-dimensional visualizations to be created and manipulated through direct manipulation of table headers and corresponding axes 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 1. 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 (fx) 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. Finally, as in a spreadsheet, our system allows fields (columns) to be hidden from view and later recalled using the context menu.

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 2.

### 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. 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 courses and displays a selection of its fields:

<table>
<thead>
<tr>
<th>id</th>
<th>area</th>
<th>title</th>
<th>may_pdf</th>
<th>may_exam_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>2</td>
<td>Roman Art</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>177</td>
<td>2</td>
<td>Comedy</td>
<td>Y</td>
<td>Final</td>
</tr>
<tr>
<td>845</td>
<td>2</td>
<td>Russian Drama</td>
<td>N</td>
<td>Other</td>
</tr>
<tr>
<td>1795</td>
<td>4</td>
<td>American Politics</td>
<td>Y</td>
<td>Y Final</td>
</tr>
<tr>
<td>2566</td>
<td>4</td>
<td>Junior Seminars</td>
<td>N</td>
<td>N Other</td>
</tr>
<tr>
<td>3921</td>
<td>4</td>
<td>Judicial Politics</td>
<td>Y</td>
<td>Y Final</td>
</tr>
</tbody>
</table>

#### 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:

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 (ม) 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 1). The one-to-many icon ( aşağıdaki) 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 (45), 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:

<table>
<thead>
<tr>
<th>courses</th>
<th>title</th>
<th>max., min., may., must, audit, type</th>
<th>author_name</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roman Art</td>
<td>N</td>
<td>Y</td>
<td>Other</td>
<td>Ramage</td>
</tr>
<tr>
<td>Comedy</td>
<td>Y</td>
<td>Y</td>
<td>Final</td>
<td>Molire</td>
</tr>
<tr>
<td>Russian Drama</td>
<td>N</td>
<td>N</td>
<td>Other</td>
<td>Pushkin</td>
</tr>
<tr>
<td>American Politics</td>
<td>Y</td>
<td>Y</td>
<td>Final</td>
<td>Rosenzweig, Gerald</td>
</tr>
<tr>
<td>Judicial Politics</td>
<td>Y</td>
<td>Y</td>
<td>Final</td>
<td>Rosenzweig, Gerald</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>courses</th>
<th>area code</th>
<th>title</th>
<th>max., min., may., must, audit, type</th>
<th>author_name</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA</td>
<td>Literature and the Arts</td>
<td>Roman Art</td>
<td>Ramage</td>
<td>German Art and Philosophy</td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>Literature and the Arts</td>
<td>Comedy</td>
<td>Molire</td>
<td>Feydeau</td>
<td>A Plea in Her Ear</td>
</tr>
<tr>
<td>LA</td>
<td>Literature and the Arts</td>
<td>Russian Drama</td>
<td>Pushkin</td>
<td>Chernev</td>
<td>The Seagull</td>
</tr>
</tbody>
</table>

Notice that tuples in the READINGS relation occur independently of tuples in the SECTIONS relation; this kind of visualization cannot be constructed in tools based on flat tabular results (see Related Work). Also notice the absence 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 (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 AUTHORITY_NAME, then by TITLE:

<table>
<thead>
<tr>
<th>courses</th>
<th>title</th>
<th>max., min., may., must, audit, type</th>
<th>author_name</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roman Drama</td>
<td>0</td>
<td>Cherkov</td>
<td>Pushkin</td>
<td>Vampilov</td>
</tr>
<tr>
<td>Junior Seminars</td>
<td>12</td>
<td>Lazarus, Edward</td>
<td>Rosenzweig, Gerald</td>
<td>The Hollow Hope</td>
</tr>
<tr>
<td>Judicial Politics</td>
<td>24</td>
<td>Lazarus, Edward</td>
<td>Rosenzweig, Gerald</td>
<td>The Hollow Hope</td>
</tr>
<tr>
<td>Roman Art</td>
<td>25</td>
<td>Gombich</td>
<td>Ramage</td>
<td>German Art and Philosophy</td>
</tr>
</tbody>
</table>

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 1), a filter can be defined on any field, indicated by the filter icon (льт.). 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 SECTIONS relation is filtered to show only tuples for which the DAY is W:

<table>
<thead>
<tr>
<th>courses</th>
<th>title</th>
<th>max., min., may., must, audit, type</th>
<th>author_name</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comedy</td>
<td>99</td>
<td>Molire</td>
<td>Feydeau</td>
<td>A Plea in Her Ear</td>
</tr>
<tr>
<td>American Politics</td>
<td>78</td>
<td>Rosenzweig, Gerald</td>
<td>The Hollow Hope</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>courses</th>
<th>title</th>
<th>max., min., may., must, audit, type</th>
<th>author_name</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roman Art</td>
<td>25</td>
<td>Ramage</td>
<td>Gombich</td>
<td>German Art and Philosophy</td>
</tr>
<tr>
<td>Comedy</td>
<td>99</td>
<td>Molire</td>
<td>Feydeau</td>
<td>A Plea in Her Ear</td>
</tr>
<tr>
<td>Russian Drama</td>
<td>0</td>
<td>Pushkin</td>
<td>Chernev</td>
<td>Vampilov</td>
</tr>
</tbody>
</table>

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 [26, 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:
The calculated field DURATION, marked with the formula icon (\(\sum\)), 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:

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.

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:

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:

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 (\([\sqcap]\)):

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\(^2\). She has compiled, in the table PLANTS_OS, a list of major ethanol producers\(^3\), 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)\(^4\).

\(^3\)Renewable Fuels Association/Maple Etanol SRL (2012)
Base table. The user starts by opening the table of ethanol producers as a template for the new query:

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 [40], AppForge [50], and App2You [31].

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:

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 (F) and then by quarter (F_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 (F) 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:

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 LOBBYING 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 (\(\square\)).

## 4. Formative User Study

We conducted a formative user study with 14 participants (de-noted A through N. 5 male, median age 42). 7 had experience with SQL, 11 used Excel daily. 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 second part of the study, and as time permitted instead, initial tasks were designed to act as training tasks for subsequent ones. In the second part of the study, and as time permitted.

**Manual joins.** Performing the lobbying query from Section 3.3, most users moved through the manual join dialog quickly and correctly on their first attempt. Still, users preferred automatic joins once introduced to them, see below. Users attempting the inflation correction portion of the query had no problems with the join against the CPI table; only users DG required a hint that they would need to use the JOIN feature again.

**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 noted 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 SIEUFEHRD 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. User G 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 a training task, users applied them quickly and accurately in a follow-up task, despite the follow-up task requiring more steps. This suggests users are able to apply formulas effectively after first learning them, but 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; automatic joins. Working on the course catalog dataset that was seen in Section 3.2, users were generally able to use the automatic foreign key 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 automatic join feature, using words such as “fantastic”, “wow”, “darn”, 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.”

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) [9]. Tullis and Stetson [47] 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). 2 had prior experience with the Access query designer, 6 had significant exposure to SQL. 2 used Excel daily, the rest weekly or monthly. We met with each user for a single study session, structured as follows:

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.
5. Work through the same tasks in Tool 2, under otherwise identical conditions. Stop after about 20 minutes.
7. Discussion and feedback.

The standardized tasks [5], 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). 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 2. The raw SUS score is reported along with separate Learnability and Usability scores as defined by Lewis and Sauro [34], as well as a percentile rating among 30 other studies in the B2B (Business Software) category as detailed by Sauro [42]. The difference in raw SUS scores between Access and SIEUFERD is statistically significant ($p = 0.0019$ with two-tailed paired t-test).

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. In future work, we hope to incorporate editing of data in our system; this will allow SIEUFERD to act as a complete schema-independent end user front-end for relational databases.
7. REFERENCES


