Expressive Query Construction through Direct Manipulation of Nested Relational Results

by

Eirik Bakke

S.M., Massachusetts Institute of Technology (2011)
B.S.E., Princeton University (2008)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

Author . . . . . .
Department of
Electrical Engineering and Computer Science
August 30, 2016

Certified by . . . . . .
David R. Karger
Professor
Thesis Supervisor

Accepted by . . . . . .
Professor Leslie A. Kolodziejski
Chair, Committee on Graduate Students

Expressive Query Construction through Direct Manipulation of Nested Relational Results

by

Eirik Bakke

Submitted to the Department of
Electrical Engineering and Computer Science
on August 30, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

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. This makes the power of relational databases largely inaccessible to non-programmers. This thesis proposes a solution, in two parts.

The first contribution of this thesis is a solution to the visual query language problem, that is, the problem of letting end users construct arbitrary database queries through a graphical user interface. We propose the first visual query language to simultaneously satisfy three requirements: (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. By directly manipulating nested relational results, and using spreadsheet idioms such as formulas and filters, the user can express arbitrary SQL-92 queries while always remaining able to track and modify the state of the complete query.

The second contribution of this thesis is an algorithm for automatically formatting nested relational data using the traditional visual idioms of hand-designed database UIs: tables, multi-column forms, and outline-style indented lists. The algorithm plugs directly into the output stage of our visual query language, and produces the concrete graphics that the user sees and manipulates on the screen during query construction. The algorithm eliminates the need for an application developer to specify low-level presentation details such as label placements, text field dimensions, table column widths, and list styles.

Our prototype visual query system gives the user an experience of responsive, incremental query building while pushing all actual query processing to the database layer. We evaluate the query building aspects of our system with formative and controlled user studies on a total of 28 spreadsheet users. The controlled study shows our system outperforming Microsoft Access by 18 points on the System Usability Scale [17]; this corresponds to a 46 percentage point difference on a percentile scale of other studies in the Business Software category. We also evaluate the different layouts that can be produced by our automatic layout algorithm, including via an online user study on 27 subjects.

Thesis Supervisor: David R. Karger
Title: Professor

Acknowledgments

This thesis could not have existed without the tremendous support, both moral and intellectual, of my dear advisor David Karger. I will miss his familiar footsteps as I now pack up my office desk and leave after eight years at MIT. Karger's close mentorship and guidance through my first publications taught me everything I didn't know I didn't know about the process of doing academic research. At the same time, as well as during the later, more independent stages of graduate work, he has remained an ardent champion of my PhD project, always eager to jump into an extended design discussion, and to promote and acknowledge the work of his students. Graduate school has been a pleasure to go through in large part thanks to Karger.

Essential to the friendly, funny, and collegial culture I experienced in and around Karger's "Haystack" group were the early haystackers and associates–those who started in the same year as me (Ted Benson, Katrina Panovich), and the Elders (Michael Bernstein, Harr Chen, Adam Marcus, "electronic" Max Van Kleek, Eugene Wu, and Sacha Zyto). That thoroughly positive culture was also to be found in the User Interface Design group, the people of which I will miss as well.

Besides Karger, I thank Rob Miller and Sam Madden for being on my committee. Rob served as a great co-advisor for my Master's project, including my first attempt at doing a user study and getting a paper published. I thank Paul Grogan and Yod Phumpong Watanaprakornkul for helping design and implement the system that became my Master's project.

I'd like to thank my officemates–Matt Coudron, Pritish Kamath, and Lea Verou–and Patrice Macaluso, for many a daily chuckle, and my Boston friends–Christian Haakonsen, Diane Haakonsen, and Angela Limoncelli–for being my friends. I thank Emilia Díaz-Struck for our many evenings working on the ethanol lobbying project, for tango, and Karger again for introducing us.

I thank Professor Barbara Liskov for the ninth floor espresso machine.

Finally, I thank my family for their enthusiasm and frequent visits, and for being there year after year as a home to which I could always return.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Querying for Non-Programmers

Emilia Díaz-Struck is an investigative journalist who, in 2012, was writing a story on lobbying in the US ethanol biofuel industry [36]. Corporations with lobbying expenditures are required by law to report all such expenditures, along with a wealth of related information, and this data is published by The Center for Responsive Politics[1] in the form of a relational database. A key goal for our journalist was to combine the data in the latter database with data from her own research, in order to answer quantitative questions relating to her story.

Relational databases do not come with a graphical user interface. They are primarily a tool for programmers, and for those in other professions who have had the time and technical inclination to learn the query language SQL. This presented a problem for our journalist: while she was well-versed in Excel, she had no experience with SQL. How, then, could she perform the various complex database queries she had in mind? As the system described in this thesis had not yet been finished, the solution was to team up with a programmer (yours truly), who would type SQL queries while the journalist stood over his shoulders asking questions of the data.

Hiring a programmer is not an elegant solution to the problem of letting non-programmers interact with databases. This thesis will present an alternative—a new kind of graphical query interface, called SIEUFERD, that has the power of SQL but the ease-of-use of a spreadsheet. We will start this thesis by showing an example interactive query building session in the SIEUFERD system, based on one of the many queries our journalist needed to construct in order to collect data for her story.

1.1.1 An Example Session in the SIEUFERD Query Builder Interface

The user (our journalist) has compiled, in the table PLANTS_OS, a list of major ethanol producers[2], 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

[1] https://www.opensecrets.org

[2] Renewable Fuels Association/Maple Etanol SRL (2012)

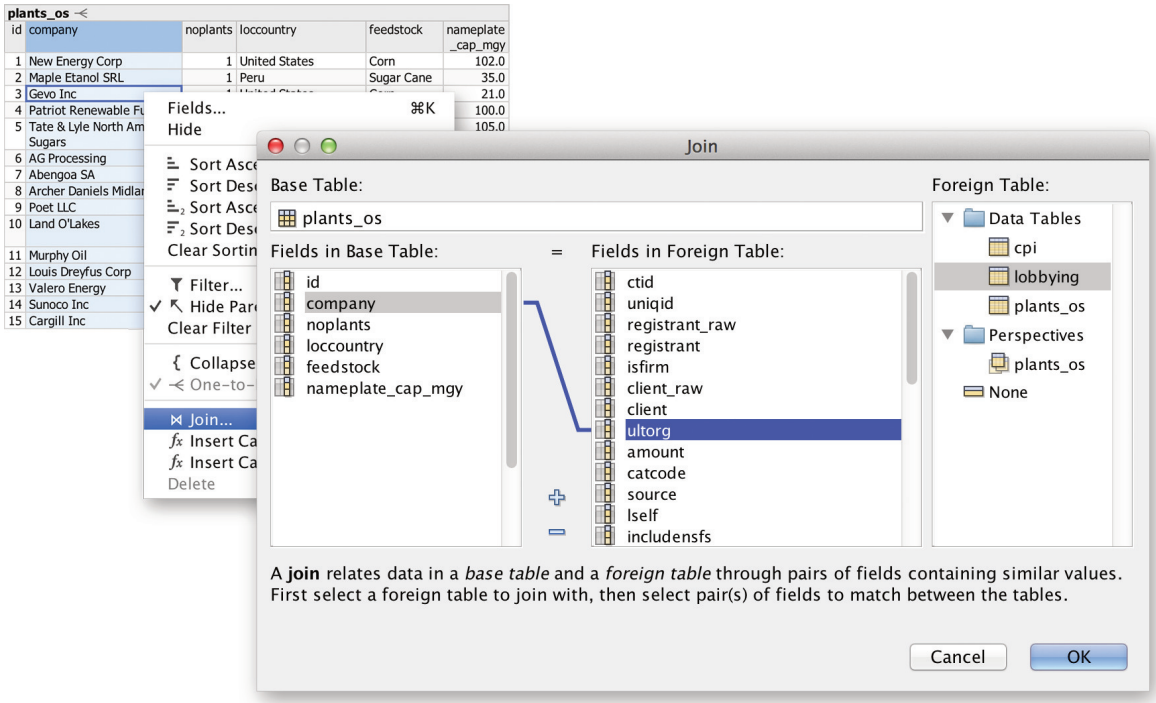
through 2012 (727,927 tuples)[3].

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

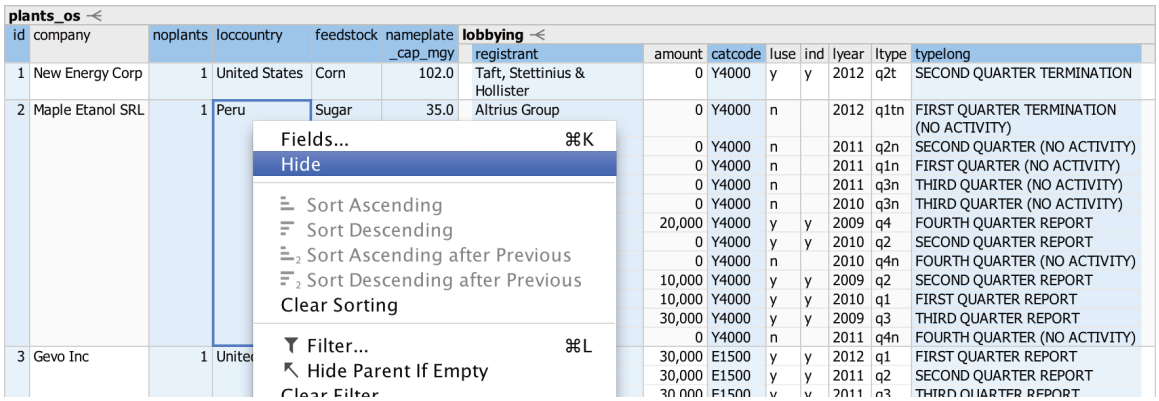
Join. To add another table to the query, the user selects the column or columns to join on and invokes the JOIN action from the context menu. This opens a dialog box for selecting the table to join with, in this case LOBBYING, and for selecting the corresponding columns from the latter to be matched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



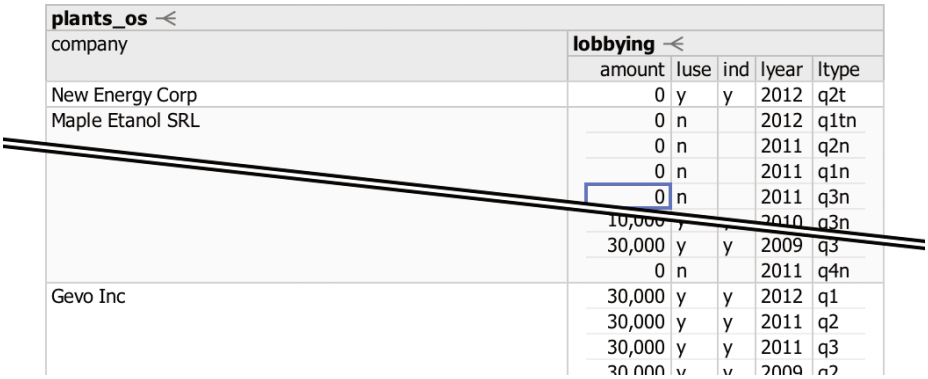
In cases where the database defines explicit foreign key relationships between tables, use of the above JOIN dialog is unnecessary; instead, all available joins are available as hidden relations in the field selector. The effect is a schema navigation capability analogous to that of QBB [84], AppForge [108], and App2You [66].

[3]The Center for Responsive Politics (2012) https://www.opensecrets.org

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



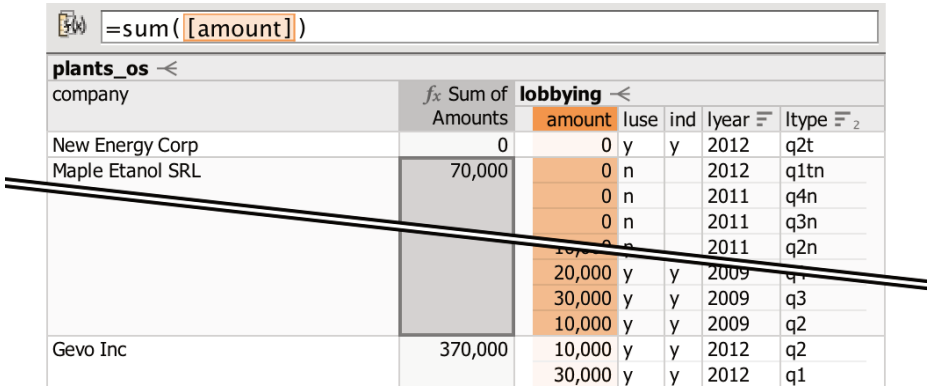
It is now easier to get a sense of the data. We have a new child relation field, called LOBBYING, containing the lobbying reports for each company:



We see the first three of the companies from the PLANTS_os table, and, for each company, their lobbying reports. The one-to-many icon (≺) on LOBBYING indicates that each company may have more than one lobbying report. The ULTORG field of the LOBBYING table, which we joined on, was automatically hidden by the JOIN action, because the equijoin constraint makes it redundant with respect to the COMPANY field.

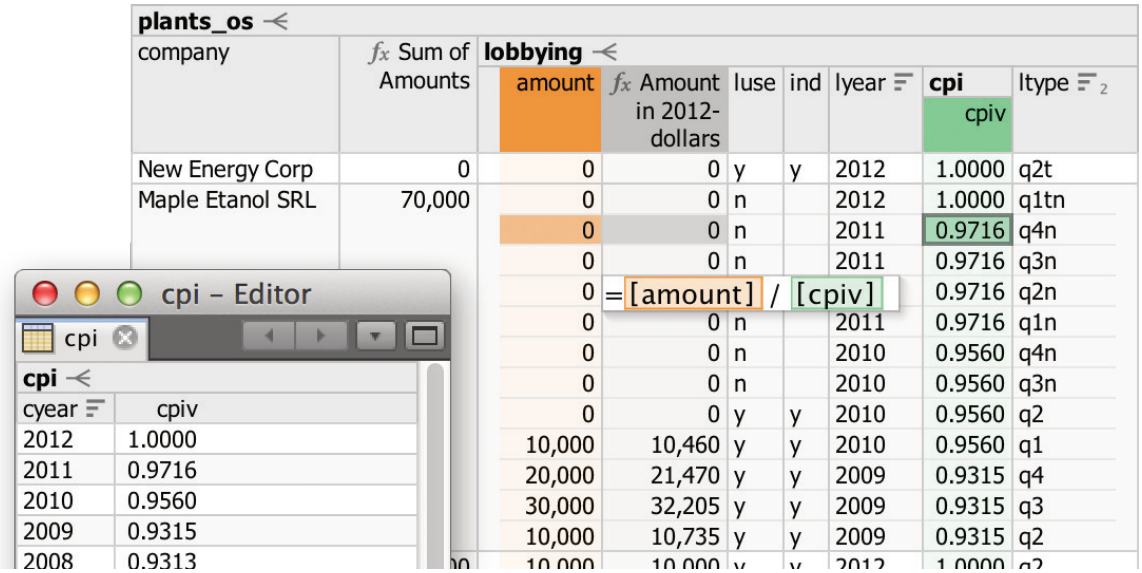
Sort. The user decides to sort the lobbying reports for each company most-recent-first, invoking the SORT DESCENDING action on the LYEAR field and then invoking the SORT DESCENDING AFTER PREVIOUS action on the LTYPE field. This sorts individual LOBBYING relations by year ($\overline{=}$) and then by quarter ($\overline{=}_2$):

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_x) 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:

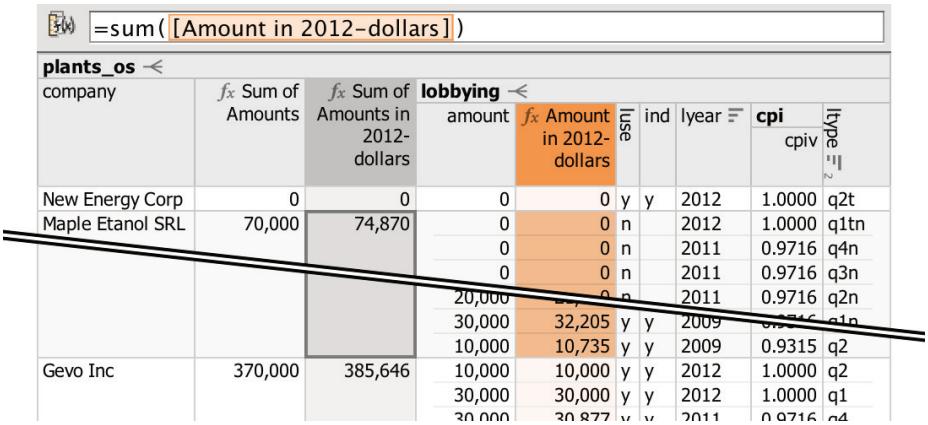


Unlike in a spreadsheet, there is no need to "drag down" the sum formula; it is always evaluated once for each tuple in PLANTS_OS, its parent relation. During formula editing, our user interface communicates the all-column behavior of formulas by highlighting the entire column of the calculated field as well as of its referenced fields. The highlight also varies, depending on the location of the cursor, to indicate which tuples contribute to a particular calculated value. As in a spreadsheet, formula references are color-coded to show the correspondence between reference tokens in the formula string and the values they reference elsewhere in the result area. New references can be inserted into the edited formula by clicking anywhere in the column of the target field, or by moving the cell cursor with the arrow keys, as in Excel. Thus, even the textual entry of arithmetic expressions can be done through some degree of direct manipulation.

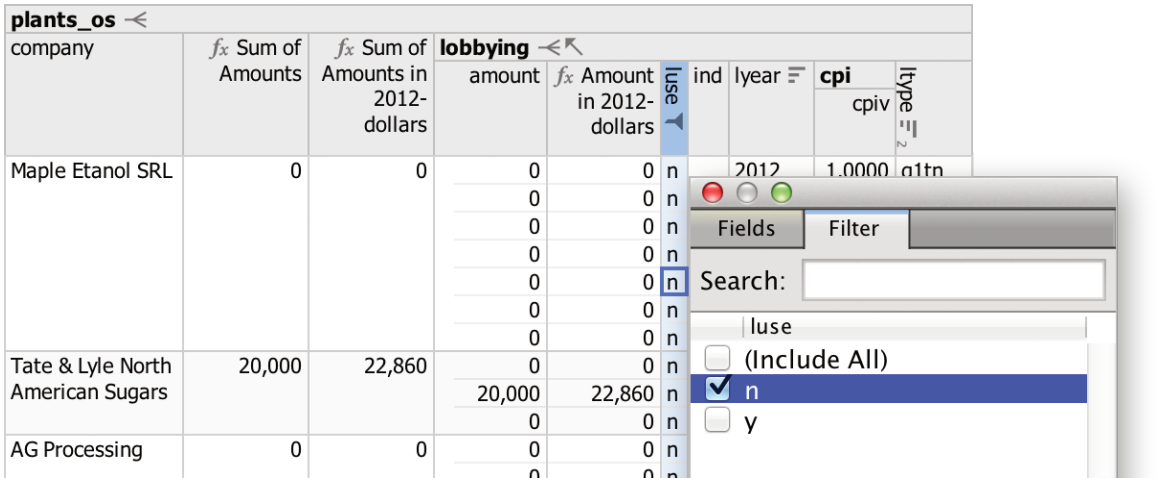
Scalar formula. Reported lobbying amounts come from different years, some going back to 1998. The user would like to calculate inflation-corrected totals. A separate table CPI contains yearly Consumer Price Index values normalized for 2012. The user performs another JOIN, this time between LOBBYING and 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:



The CPI relation does not display the one-to-many icon (⋖), as this relation was joined on its instantiated table's primary key and our system thus deduced that at most a single tuple would exist in CPI for each parent tuple in LOBBYING. 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:

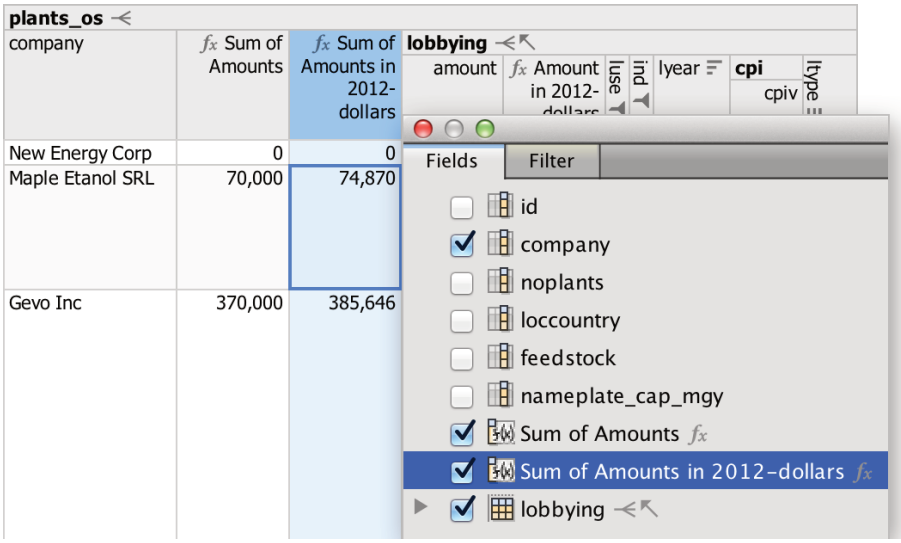


Filter. Lobbying reports may sometimes be amended, in which case the superseded reports should be excluded from totals to avoid double counting. The user can look for superseded reports by invoking the FILTER action on the LUSE field and selecting the value N:



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:

Lobbying by Ethanol Producers

Company	Plants	Feedstock	Sum of Amounts in 2012-dollars
Cargill Inc	2	Corn	16,725,489
Sunoco Inc	1	Corn	15,277,872
Archer Daniels Midland	8	Corn	9,277,472
Murphy Oil	1	Corn	7,729,618
Valero Energy	10	Corn	7,047,974
Land O'Lakes	1	Cheese Whey	4,821,907
Poet LLC	27	Corn	3,769,377
Louis Dreyfus Corp	2	Corn	2,310,378
Tate & Lyle North American Sugars	1	Corn	2,204,061
Abengoa SA	6	Corn	1,585,509
Gevo Inc	1	Corn	385,646
AG Processing	1	Corn	365,947
Patriot Renewable Fuels	1	Corn	135,215
Maple Etanol SRL	1	Sugar Cane	74,870
New Energy Corp	1	Corn	0

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 (⬚).

We have just shown how a complex database query can be constructed entirely by interacting directly with the data in the database, much like in a spreadsheet. As will be shown in user studies (Chapter 3), the concepts involved can be learned in under an hour, making our tool a realistic alternative to learning SQL for non-programmers like our journalist.

1.2 Background

Modern relational database management systems (*relational databases*), embodied in commercial products such as Oracle, IBM DB2, and Microsoft SQL Server, as well as open source projects such as PostgreSQL, MySQL, and SQLite, owe their heritage to Edgar Codd's *relational data model* [33] from 1970 and the subsequent early implementations INGRES [48] at Berkeley and System R [7] at IBM Research, both in development by 1975. In 1984, the Great Debate over data models that had raged in academic circles was settled, by decree of IBM, in favor of the relational model and the query language SQL [101]. Three decades later, relational databases are now firmly established as a universal backend for persistence and query processing, with SQL fulfilling the role, in the words of Michael Stonebraker, as "intergalactic data-speak" [35].

The persistence and query processing facilities provided by a relational database form only the bottom level of the software application stack. From the earliest days of relational database research, it was recognized that textual query languages such as QUEL and SEQUEL (later SQL), while useful in application development, would not by themselves serve as an effective user interface for non-programmers. Starting with CUPID [75] and Query-by-Example [111], again from Berkeley and IBM Research in the mid-70s, the subfield of *visual query systems* began to grow in parallel with the more storage- and

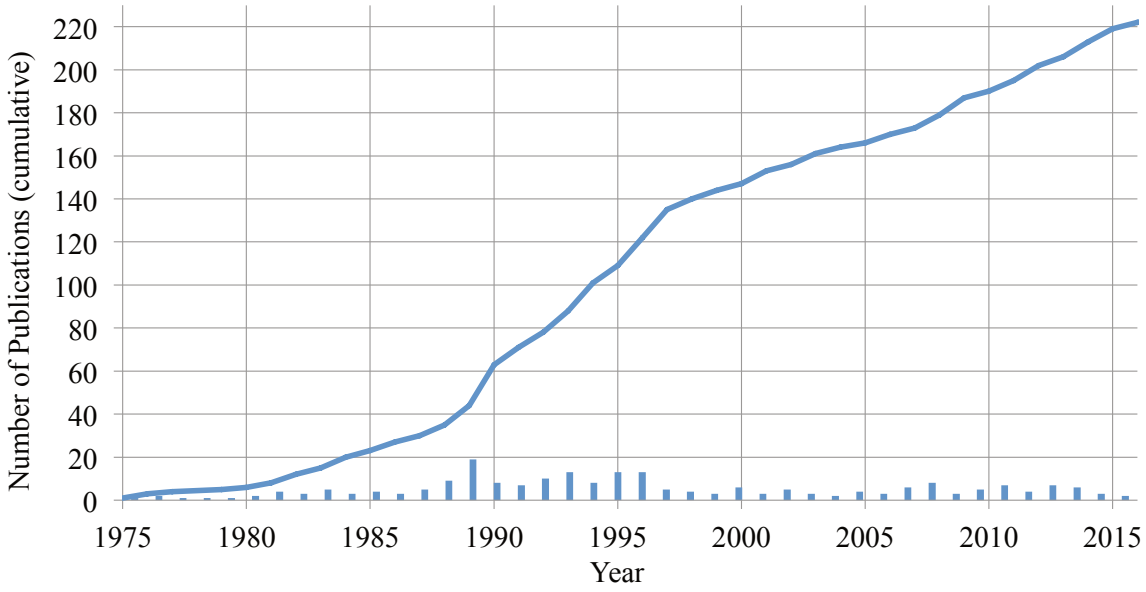


Figure 1-1: An estimate of the number of publications in the research area of visual query systems, since 1975. The vertical bars show the number of papers per year, while the line shows the cumulative number of papers after each year. Based on a subset of references in surveys by Catarci et al. [22], El-Mahgary and Soisalon-Soininen [37], and Bakke and Karger [9].

performance-oriented research on relational databases. Whereas a relational database serves as an application's *back-end*, a visual query system is concerned with its *front-end*, that is, the user interface (UI). The purpose of a visual query system is to provide end users with a friendly, graphical way to express database queries, ideally permitting the construction of any query that can be expressed in the underlying textual language (e.g. SQL). Researchers also use the term *visual query language* to refer to the specific visual grammar and semantics that is implemented by a visual query system [22].

Since, today, any off-the-shelf relational database can be used as a backend for new applications, relational databases can be considered a solved research problem. The same can not be said about visual query systems. Four decades after Query-by-Example, 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 [67, 56, 8]. Queries that involve "complex aggregates, nesting, correlation, and several other features remain on a tall pedestal approachable only by the initiated" [51]. 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 [10].

The absence of a definitive solution to the visual query language problem is not for a lack of research. Since 1998, a new paper has been published on the topic roughly every 80 days on average. The preceding decade saw even more publications, accompanying the general rise of Graphical User Interfaces in industry. See our informal tally in Figure 1-

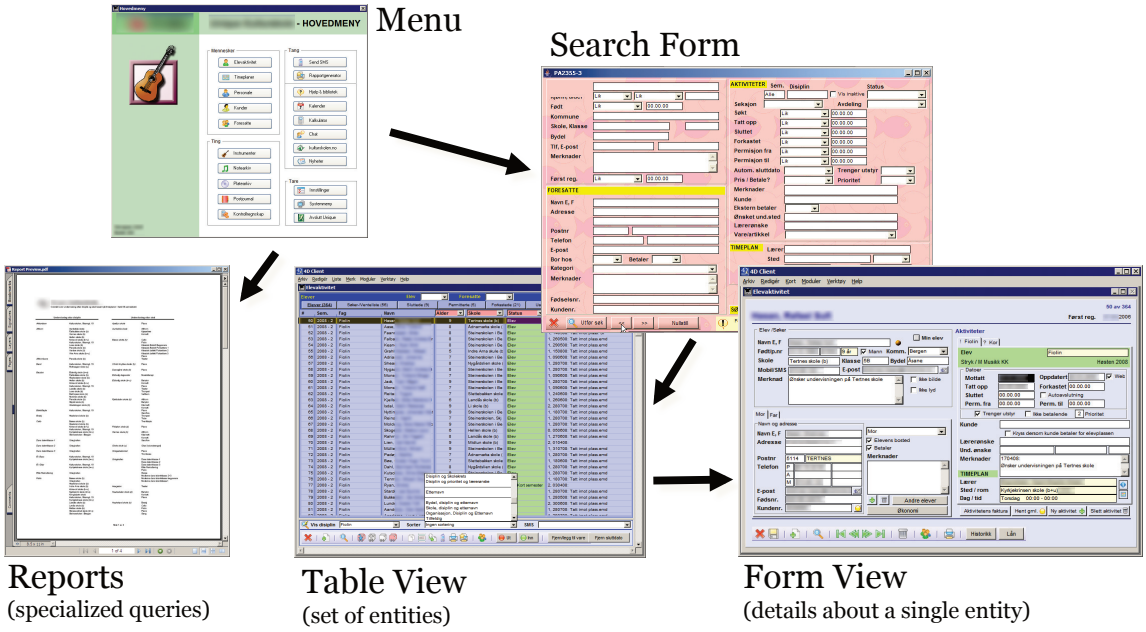


Figure 1-2: The stereotypical user interface of a tailored CRUD-style database application. Screenshots from an administration system for public Norwegian music schools.

1. The current state of database usability is described well by Jagadish et al. [54] as well as in interview studies done among business analysts [61] and nonprofits [106]. The 2016 Beckman Report on Database Research affirms the continued need for a query interface that allows users to consume data without resorting to SQL [1]. This thesis aims to solve the latter problem. Before explaining our specific approach, we will discuss the main classes of graphical database user interfaces in use today.

1.3 User Interfaces for Databases

1.3.1 Tailored CRUD Applications

A large class of domain-specific software applications serves chiefly to provide a graphical user interface to some underlying relational database. These applications allow the user to perform basic Create, Read, Update, and Delete (CRUD) operations on data in the database as well as perform a range of pre-selected query and reporting tasks. While the particular user interfaces of CRUD-style database applications invariably differ from one schema to another, their basic structure have remained the same since the early days of graphical user interfaces. This structure, popularized by developer tools such as 4th Dimension[4] (1984), FileMaker[5] (1985), and Microsoft Access[6] (1992), is illustrated in Figure 1-2. The user retrieves records using a *search form*, views results in a *table view*, and edits or views the

[4] http://www.4d.com
[5] http://www.filemaker.com
[6] http://office.microsoft.com/access

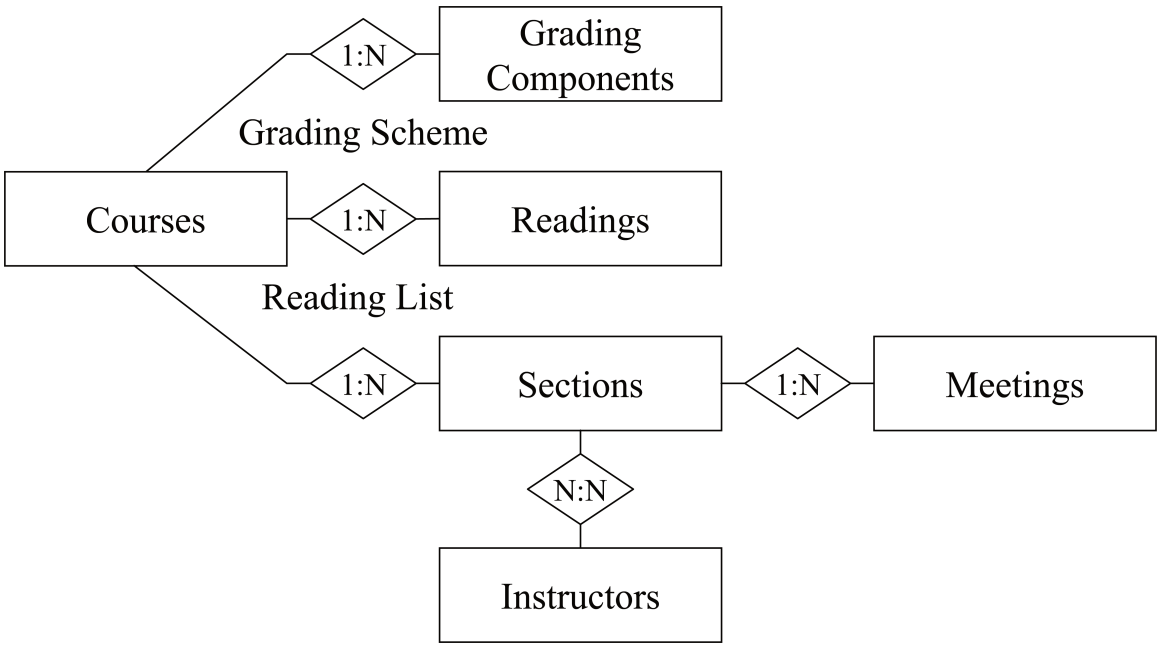


Figure 1-3: Entity-Relationship diagram of the schema for an academic course management database.

details of individual entities in a *form view*. Queries that cannot be expressed through the use of a search form are instead made available as read-only *reports*, which are hard-coded by the application developer.

A typical relational database *schema* defines numerous tables (*relations*) in order to model real-world entities and the relationships between them. An example is an academic course management system shown in Figure 1-3, illustrated using an Entity-Relationship (ER) diagram [26]. Here, each COURSE can have any number of READINGS in its READING LIST, and be associated with any number of SECTIONS (lectures, recitations, etc.). Each SECTION can be associated in turn with any number of INSTRUCTORS or MEETINGS. We call these *one-to-many* relationships. In addition to each SECTION being associated with any number of INSTRUCTORS, each INSTRUCTOR can be associated with any number of SECTIONS. A relationship that is one-to-many in both directions, like this one, is known as *many-to-many*. A typical relational database has one table per entity set plus one extra table per many-to-many relationship set (e.g. INSTRUCTOR_SECTION_ASSIGNMENTS). This organization prevents data from being stored redundantly, following the rules of relational database normalization [64].

Two aspects of the standard CRUD-style user interfaces are crucial for allowing the user to manage relationships between entities in the database. First, they can provide the user with many different views of the data, with each view potentially combining data from multiple tables in the database. Second, these interfaces are not restricted to simple tabular views, but can expose relationships in a *nested* fashion. For instance, a form view for a single entity can itself contain tab rows or miniature table views to represent multiple, independent sets of entities related to the main entity through one-to-many relationships. Note that in tailored, domain-specific applications, views are hard-coded by a developer

for the particular schema in question. Continuing the course catalog example, a tailored CRUD interface might expose many different views of the underlying data, such as "a list of sections by instructors, each section showing its associated course," or "a list of courses, each course showing its reading list, grading scheme, and sections, each section showing its meetings and assigned instructors."

The data in a nested view can be represented in any tree-structured data model; common ones include the *nested relational data model* [53, 68], XML, and JSON. Korth and Roth [65] provide an early explanation of why nested relations are a good data model for representing forms in a database application.

While tailored database user interfaces may do their job and, in fact, be of great value to their target user base, they have a number of drawbacks compared to more general-purpose software:

- The software development costs per user is high, since the target market for any given domain-specific schema is small.
- Since fewer development resources are available, tailored applications seldom reach the same level of functional maturity as more general-purpose ones. Features that are taken for granted in general-purpose applications may never be implemented for tailored ones, because the development time would not be justified for the size of the user base. Examples include undo/redo, keyboard shortcuts, drag-and-drop, accessibility, and support for international character sets.
- Complex applications with smaller deployments and fewer developers are likely have more unreported and open bugs.
- Tailored applications require users to go through a new learning period, and may require expensive training.
- With a gap between users and developers, and with application and user interface code that is tightly integrated with the particular structure of the database schema, users are not fully in control of their data. It is difficult or impossible for end-users to import data from or export data to other sources, and changing the schema in even minor ways can be a major undertaking.

1.3.2 Spreadsheets

When the effort required to either adopt or develop a new domain-specific database application is too high, information workers instead turn to a general and more familiar tool: the spreadsheet. One survey [82] shows that "sorting and database facilities" are the most commonly used spreadsheet features, with 70% of business professionals using them on a frequent or occasional basis. In contrast, less than half use "tabulation and summary measures such as averages/totals"—one of the design goals of the original VisiCalc spreadsheet—or more advanced features. Furthermore, spreadsheet users "shun enterprise solutions" [87] and "do not appear inclined to use other software packages for their tasks, even if these

packages might be more suitable" [24]. "Export to Excel", the joke goes, "is the third most common button in data and business intelligence apps... after OK and Cancel"[7].

Shneiderman [92] 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." The concept of direct manipulation is central to the visual query system presented in this thesis. The direct manipulation aspects of our own interface will be discussed in detail in Chapter 3.

Besides its nature as a direct manipulation interface, the spreadsheet, and Microsoft Excel in particular, affords a large range of streamlined facilities for working with any data that can be arranged in a grid of cells, including multiple selection, copy/paste, find/replace, undo/redo, inserting and deleting, extending data values, sorting and filtering on arbitrary fields, navigating and selecting cells with the keyboard, and so on. For reasons explained before, tailored database user interfaces seldom reach this level of sophistication. When it comes to general editing tasks on tabular data, spreadsheet systems have an advantage even over most tailored applications.

While spreadsheets are great for managing data which can naturally be modeled as a single table of data, they are less than ideal for database tasks, which tend to involve a multitude of entity types and relationships between them. Because traditional spreadsheet UIs provide no easy way to create joined views of related tables, it becomes impractical to follow good practices of schema normalization, which call for tables of redundantly represented data to be decomposed into multiple smaller ones. This in turn exposes all the usual problems associated with managing improperly normalized data, i.e. insertion, update, and deletion anomalies [64].

We previously mentioned the importance of nested views as a way to present relationships between entities in tailored database user interfaces. Nested views can often be represented in spreadsheets using clever formatting tricks, e.g. indentation, skipped cells, or comma-separated lists. This strategy does not scale well. Besides being hard to generalize, such views would have to be manually created and, if more than one view of the same data is desired, kept in sync with the original data. This is infeasible if data changes often, if there are many views, or there are multiple users. Furthermore, when the data representation deviates from a simple tabular format, key spreadsheet features such as sorting, inserting, charting, filtering, or even simple navigation between individual units of data become hard or impossible to apply correctly. A recent empirical study [89] discusses additional problems that arise when organizations use spreadsheets as information systems. Extensive literature also exists on the broader topic of errors in spreadsheets [85, 80].

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

1.3.3 Business Intelligence Tools

A number of visual query systems exist in the commercial marketplace, often marketed under the umbrella term of *Business Intelligence* (BI) software. A survey of such products is done regularly by Gartner[8]. Unlike tailored CRUD applications, BI tools tends to be designed as general-purpose software that can be made to work with any existing database instance. And unlike spreadsheets, BI tools let the user define new queries on existing data, without altering the underlying data. In fact, BI tools are usually considered read-only, and thus fulfill different organizational needs than CRUD applications.

Subcategories of visual query systems in the Business Intelligence category include *report generators*, focused on batch creation of custom-formatted documents from data in a database, and *visualization* tools, focused on interactive exploration of data through diagrams and plotted graphics. Report generator products have been surveyed by Król [67]; examples include Crystal Reports[9] and Cognos[10]. Visualization products include Spotfire[11] and Tableau[12], both which sprung out of their respective founders' PhD projects [5, 4, 98, 100].

While report generators may provide facilities for producing graphs and charts, their primary use case is for text-based data presentations. Even in Król's case study, which was done in the context of a geographical information system, only 3 out of 117 reports contained a map visualization, with those reports receiving little actual usage [67]. That said, report generators can typically be used to produce a broader class of text-based data representations than visualization tools, including support for elements such as tables, forms, paragraphs, and bullet lists. These are the same kinds of visual layouts that are dominant in tailored CRUD applications, and developers of the latter will often use report generators to develop read-only portions of the application. Note that the generation of layouts in a report generator is seldom fully automatic. A developer using a report generator to produce complex form-like views, for instance, will spend a significant amount of time dealing with low-level presentation details such as label placements, text field dimensions, table column widths, and list styles. One contribution of this thesis is an algorithm that fully automates the design of such layouts.

We have yet to see a Business Intelligence product that successfully enables even advanced spreadsheet users to express arbitrary database queries without learning SQL. Like Liu and Jagadish [71], we believe this to be due to the hard user interface design problem of providing high expressiveness without violating the rules of direct manipulation. In particular, previous direct manipulation systems either sacrifice expressiveness or hide the actual query from the user. We will examine and solve this problem in Chapter 3.

[8] Gartner's Magic Quadrant for Business Intelligence and Analytics Platforms
https://www.gartner.com/doc/reprints?id=1-2XXET8P&ct=160204

[9] http://www.crystalreports.com

[10] http://www.ibm.com/analytics/us/en/technology/cognos-software

[11] http://spotfire.tibco.com

[12] http://www.tableau.com

1.4 Contributions

1.4.1 A Visual Query Language

The first contribution of this thesis is a solution to the visual query language problem. Following the requirements articulated by Liu and Jagadish [71], we present the first visual query language to support both the specification and subsequent modification of arbitrary SQL queries from within a pure direct manipulation interface. Specifically, our visual language is the first to meet all of the following requirements:

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 progressively refined visual query [71].

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. [71]

R3. SQL-like expressiveness from within the direct manipulation interface. A minimum requirement is for the visual language to be relationally complete [32] as well as to support aggregation in arbitrary multi-block queries. Our own visual language covers a larger set of operators, sufficient to express any `SELECT` statement valid in SQL-92.

Our solution is based on the idea of combining the query and its result into a single spreadsheet-like visual representation. Liu and Jagadish attempted a similar approach, but encountered several problems—"points of non-commutativity", "where to store and display the result of [aggregations]", and limited expressiveness (single-block queries only). We believe that such problems are a consequence of attempting to map query expressions, which may be nested in nature, onto a flat result set. For instance, SQL queries may contain arbitrarily nested `SELECT` clauses, performing new aggregations at each level, even though the final output is always a simple flat table of results.

Our own visual query language solves the problem of mapping query expressions to query results by extending the data model of results such that queries may return *nested* results. Specifically, we allow queries to produce results from the *nested relational data model* [53, 68]. 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. In the ethanol lobbying example, the user could choose to see not only the total lobbying expenditures for each company, but also the complete list of lobbying reports that contributed to each total, including the inflation correction calculations that were done at the level of each individual report, and the filters that were used to exclude certain reports.

In our user interface, the user will always be looking at the nested relational result of the visual query currently being built, formatted using a nested table layout. See Figure 1-4. All query manipulation actions are initiated from the result layout, satisfying requirement R1. In a nested table layout, the table's header area visually encodes the schema of the

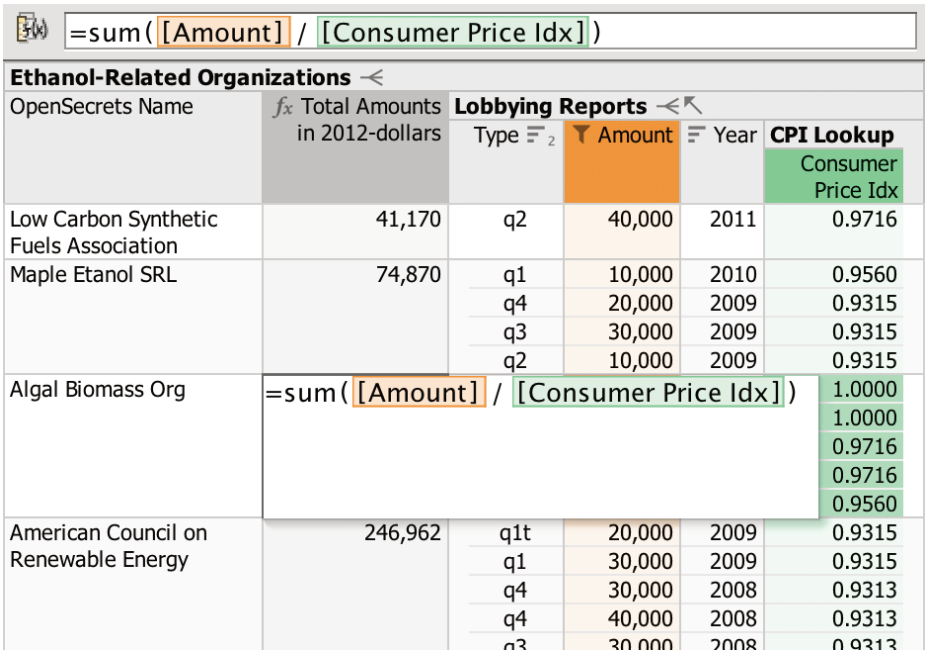


Figure 1-4: Visual specification of a database query in a spreadsheet-like environment. The query structure is encoded in the table header, which shows three joined table instances (bold labels), one-to-many relationships (≺), sorting (), active filters (, ↖), and a formula (*fx*). Formulas can be edited either directly in cells or through a formula bar.

nested result, including which fields are nested under others in the nested data model. 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. The user can then manipulate any part of the query by initiating an action on the corresponding field in the result layout, satisfying requirement R2. 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, satisfying requirement R3, while always remaining able to track and modify the state of the complete query.

1.4.2 Automatic Formatting of Query Results

The second contribution of this thesis is an algorithm for automatically formatting nested relational data into table-, form-, and report-style layouts. The algorithm plugs directly into the output stage of our visual query language, and produces the concrete graphics that the user sees and manipulates on the screen during query construction. The combination of our visual query language with our automatic layout algorithm yields a powerful direct manipulation interface in which the user can quickly produce any of the views typically found in tailored CRUD applications. This eliminates the need for an application developer to specify low-level presentation details such as label placements, text field dimensions, table column widths, and list styles, as well as the need for custom code for populating each view with data from the database. Furthermore, generated layouts automatically support

interaction features such as multiple selection, cursor movement by mouse or keyboard, "frozen" headers during scrolling, context menus, in-place editing of formulas and field labels, undo/redo, error highlighting, and infinite scrolling. Each of these features would require a significant development effort if implemented in the context of a tailored application.

Layouts produced by our formatting algorithm are hybrids between two existing types of layouts: outline layouts and nested table layouts. See Figure 1-5. The outline layout stacks both tuples and fields vertically in an indented bullet list. The basic hybrid layout replaces the outline layout with a nested table layout for specific relation fields in the schema, wherever such replacement can be done without making the layout too wide for the available screen or page size. This leads to more compact layouts without introducing horizontal scrolling. Finally, our algorithm reclaims additional wasted space by allowing narrow fields in the outline layout to be stacked in columns. The ideal placement of column breaks, as well as the decision to use an outline or table layout for a given relation field, is done using an idealized layout produced using average lengths of each field.

1.4.3 Prototype Implementation

Our prototype visual query system, called SIEUFERD (pronounced *soy-fird*), gives the user an experience of responsive, incremental query building while pushing all actual query processing to the database layer. We evaluate the query building aspects of our system with formative and controlled user studies on a total of 28 spreadsheet users. The controlled study shows our system outperforming Microsoft Access by 18 points on the System Usability Scale [17]; this corresponds to a 46 percentage point difference on a percentile scale of other studies in the Business Software category. We also evaluate the different layouts that can be produced by our automatic layout algorithm, including via an online user study on 27 subjects.

In our current implementation, all views generated by our system are read-only. In the future, we hope to incorporate *editing* of data; the semantics of our visual query language are already well-suited for producing updatable views. This would allow SIEUFERD to act as a general-purpose (*schema-independent*) replacement for tailored CRUD applications.

1.5 Thesis Organization

Chapter 2 reviews previous work related to the two main contributions of this thesis, our visual query language and the algorithm for automatic formatting of query results. We evaluate the most relevant past systems according to the three requirements for visual query languages that were laid out in the previous section.

Chapter 3 presents our visual query language and the specific structure of queries in our language, defined by the SIEUFERD *query model*. We describe two user studies done to evaluate the usability of our system.

Chapter 4 describes the internal representation of the SIEUFERD query model, and the mapping of operations in the user interface to modifications on the query model. We show how SQL queries are generated from an instance of the query model. We demonstrate

- Course Listings

Dept.	Number
GEO	207
AST	207

Max. Enrollment	140
May Audit?	Y
Website	
Final Exam Type	Final

Title: A Guided Tour of the Solar System

Description: This course examines the major bodies of our solar system, emphasizing their surface features, internal structures, and atmospheres. Topics include the origin of the solar system, habitability of planets, and role of impacts in planetary evolution. Terrestrial and giant planets will be studied as well as satellites, comets, and asteroids. Recent discoveries from planetary missions are emphasized. The course is aimed primarily at non-science majors.

Grading:

Title	Perc.
MidTerm Exam	20
Quizzes	10
Final Exam	20
Precept Participation	10
Other Exam	20
Problem Set(s)	20

Sample Reading List:

Author Name	Title
Morrow and Owen	The Planetary System
Bennett et al.	The Cosmic Perspective
Consolmagno and Schaefer	Worlds Apart: A Textbook in Planetary Science
Beatty et al.	The New Solar System

Sections:

Format	Number	Meetings				Instructors		
		Beg. Time	End Time	Place	Days / Day	First	Middle	Last
L	01	11:00:00	11:50:00	GUYOT 10	M W F	Thomas	S.	Duffy
P	01	13:30:00	14:20:00	GUYOT 155	T	Thomas Nicole Mark	S. K A.	Duffy Gotberg Miller
P	03	15:30:00	16:20:00	GUYOT 154	W	Thomas Nicole Mark	S. K A.	Duffy Gotberg Miller
P	04	19:00:00	19:50:00	GUYOT 154	W	Nicole Mark	K A.	Gotberg Miller
P	05	11:00:00	11:50:00	GUYOT 154	Th	Nicole Mark	K A.	Gotberg Miller

- Course Listings

Dept.	Number
CEE	471
GEO	471

Max. Enrollment	58
May Audit?	Y
Website	
Final Exam Type	Other

Title: Introduction to Water Pollution Technology

Description: An introduction to the science of water quality management and pollution control in natural systems; fundamentals of biological and chemical transformations in natural waters; indentification of sources of pollution; water and wastewater treatment methods; fundamentals of water quality modeling.

Grading:

Title	Perc.
Design Project(s)	33
Quizzes	66
Other (See Instructor)	1

Sample Reading List:

Author Name	Title
Tchobanoglous & Schroeder	Water Quality
Eckenfelder	Principles of Water Quality Management
Metcalf & Eddy	Wastewater Engineering
Sawyer & McCarty	Chemistry for Environmental Engineers

Sections:

Format	Number	Meetings				Instructors		
		Beg. Time	End Time	Place	Days / Day	First	Middle	Last
L	01	09:30:00	10:50:00	FRIEN 008	T Th	Peter Jeffery	R. Scott	Jaffe Paull

Figure 1-5: Example hybrid outline/table layout produced by our layout generation algorithm. Magnification of Figure 5-1(e).

the expressiveness of the query model by defining a translation from an arbitrary SQL-92 query, via an extended relational algebra, to a corresponding query in the SIEUFERD query model.

Chapter 5 presents our algorithm for automatic formatting of query results from our visual query system into table-, form-, and report-style output displays. We evaluate our algorithm based on the space efficiency of output layouts, on performance, and on readability.

Chapter 6 discusses the extent to which our system solves the problems mentioned in this introduction, discusses future work, and summarizes our contributions.

This thesis includes material previously published at SIGMOD '16 [9] (Chapters 2 and 3), InfoVis '13 [9] (Chapters 2 and 5), and for this introduction, CHI '11 [10] and CIDR '11 [8]. All thesis content is by the author.

Chapter 2

Related Work

We now review previous work related to the two main contributions of this thesis, our visual query language and the algorithm for automatic formatting of query results.

2.1 Visual Query Systems

Visual query systems have been surveyed by Catarci et al. [22] and, recently, El-Mahgary and Soisalon-Soininen [37]. 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 2.1 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 2.1 also indicates which systems support nested results, i.e. a graphical equivalent of a nested 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" [11]. Systems that base their result representation on a single flat table of primitive values, such as Tableau [99], are unable to express such queries. The same tends to hold for any system that takes its input from a single select-project-join query, since *multivalued dependencies* [39] 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 [25], support a restricted form of nesting where an otherwise flat result table, or a finite set of such, is displayed such that each column is grouped by values in the column immediately to the left of itself. This still does not handle PARTS↠SUPPLIERS/ORDERS-type queries from the example above. Some systems, like Rhizomer [18] and Etable [60], can display parallel one-to-many relationships, but only at a single level of nesting. Besides their use in visual query systems, nested data models have been used both in optimization [97, 20] and expressiveness analysis [70] of query languages with aggregate functions.

Tableau, as well as other systems based on the pivot table concept, produce *cross-tabulated* rather than nested results. Cross-tabulation and nesting are orthogonal concepts, in the sense that a system can support one with or without supporting the other.

Table 2.1: 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.

Direct Manip.	*Query Representation*	*Year*	*System*		*R1*	*R2*	*R3*	*Unrestricted Nested Results*
Yes	Overlaid on Result	2016	VISAGE	[83]	X	X		X
		2016	Etable	[60]	X	X		
		2014	GBXT	[3]	X	X		X
		2013	Rhizomer	[18]	X	X		
		2012	DataPlay	[2]	X	X		X
		2006	Tabulator	[14]	X	X		X
		2002	Polaris (Tableau)	[99]	X	X		
	Spreadsheet Formulas	2016	Object Spreadsheets	[74]	X	X		X
		2010	Spreadsheets as DB	[104]	X	X		
		2005	A1	[63]	X	X		X
		1997	OOF Spreadsheets	[31]	X	X		X
		1994	Forms/3	[19]	X	X		
	Exposed Algebraic	2013	Mashroom	[47]	X		X	X
		2011	Wrangler	[62]	X		X	
		1991	TableTalk	[38]	X		X	X
	Hidden Algebraic	2016	Gneiss	[25]	X		X	
		2013	GestureDB	[78]	X		X	
		2010	CRIUS	[86]	X			X
		2009	SheetMusiq	[71]	X			
		2008	AppForge	[108]	X			X
		1989	R^2	[50]	X		X	X
No	Diagram-based	2016	OptiqueVQS	[94]				
		2014	VisualTPL	[27]				X
		2011	Related Worksheets	[10]				X
		2009	App2You	[66]				X
		2005	QBB	[84]				
		2002	QURSED	[81]				X
		1990	QBD	[6]				
	Form-based	2008	Form Customization	[57]				
		1998	QBEN	[72]				X
		1997	ESCHER	[107]				X
		1989	PERPLEX	[95]				
		1977	QBE	[111]				

The difference between the two concepts can be more clearly understood by reading Section 5.4.3, where we show how to combine both cross-tabulations and nesting in a single system. To our knowledge, past systems have only supported one or the other, or neither.

Returning to our taxonomy of visual query systems, we first discuss systems that do not fall in the direct manipulation category. *Form-based* systems originated with Query-by-Example (QBE) [111], where the user populates a set of empty *skeleton tables* with conditions, variables (*examples*), and output indications. ESCHER [107] and QBEN [72] extend QBE to support nested results, while PERPLEX [95] 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 [57] 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) [6], Query-by-Browsing (QBB) [84], QURSED [81], App2You [66], Related Worksheets [10] and OptiqueVQS [94], or through a diagrammatic query plan, as in VisualTPL [27]. 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). The system from the author's own Master's thesis, Related Worksheet, falls in the same category. Its limited query-related functionality was available only available from a sidebar, disconnected from the data in the center of the screen. An exception was the TELEPORT feature, which operated on the currently selected cell.

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" [71] (requirement R2). For example, in GestureDB [78], the user has no way to modify a filter on a column that was subsequently totaled in an aggregation, since the column that was originally associated with the filter is no longer on the screen to be manipulated. Similar problems exist in R^2 [50], AppForge [108], CRIUS [86], and Gneiss [25]. SheetMusiq [71] 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 [62], the XQuery-like *mashup scripts* of Mashroom [47], or the diagram-based representation in TableTalk [38]. 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 [104] 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 [19], Object Oriented Functional Spreadsheets [31], A1 [63], and Object Spreadsheets [74]. Tyszkiewicz, in particular, intends formulas to be generated from SQL statements rather than to be typed manually by the user.

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 that current such representations are not expressive enough to support arbitrary queries (requirement R3). For example, the direct manipulation interfaces of Tabulator [14], Rhizomer [18], GBXT [3], Etable [60], and VISAGE [83] support filters and joins over schema relationships, but are unable to express calculation, aggregation, general-purpose joins, or other binary operators. In DataPlay [2], direct manipulation is used only to choose between universal and existential qualifiers. Tableau [99] 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.

2.2 Structured Data Visualization

An important aspect of a visual query system is the approach it takes to visualize the data that is returned from constructed queries. This is especially important for direct manipulation systems, where manipulation of the visual representation of returned data is the chief way of interacting with queries. We here discuss prior work that relates specifically to the output engine of our visual query system. The related systems discussed in this section are not necessarily all *query* systems, but may pertain more generally to the field of *information visualization*.

2.2.1 Tree Visualization

As previously mentioned, the queries constructed in our visual query system always return results in the nested relational data model, a type of tree-structured data model. Our system for automatic formatting of query results thus falls in the research area known as *tree visualization*. This area has been surveyed by Shneiderman [93] and, more recently, Graham and Kennedy [45].

In our case, we are dealing with the subproblem of *single tree* visualization, with the additional constraint that we are working with structured data only, i.e. data that conforms naturally to some schema, such as most JSON or XML data. Our visual layouts furthermore make the most sense in cases where the schema of the tree-structured data to be visualized is *non-recursive*, that is, where the schema defines a tree rather than a graph of permissible fields. This was the case for 9 out of the 10 example datasets in the XML corpus we used for our user study.

Our system's assumption that input data conforms to a schema is important for the quality of the layouts produced, because it allows us to take advantage of the resulting regularity of the data to be presented. For instance, a list of tuples with similar fields in each tuple is best presented in a tabular layout, with each column representing one field name. Analogous data model specialization is seen in some other systems, for instance Robertson's polyarchies [90].

In Graham's taxonomy, the visualizations produced by our system form a hybrid between the *nested* and *indented list* representations. While nested tables, an important base case in our layout system, organize data in a grid of rows and columns, they do not fall into the *matrix* representation category, but rather among the nested representations, since data leaf nodes are always contained within the visual boundaries of their parent nodes.

The study by Chimera and Shneiderman [29] compared three variations of the outline-style indented list view; two of these were interactive. Their results suggest that future versions of our system should include collapsible nodes in indented lists. Ziemkiewicz et al. [110] compare four different classes of tree visualizations, including three nested variants. None of the layouts tested include tables or hybrids between tables and indented list representations. Ghoniem et al. [44] provide a taxonomy of readability tasks on graphs and present a study comparing node-link and matrix representations.

Treemap [59] is maybe the most well-known nested or Venn Diagram-style tree representation. Treemaps fit an arbitrarily sized tree into a viewport of pre-defined dimensions, by making subsequent levels smaller and smaller. The relative size of siblings is determined by some semantically significant weight associated with each node in the tree, such as disk space consumed in a map of a file system hierarchy. In contrast, our own system allocates to leaf nodes whatever visual area is necessary to display the contained primitive text values at a constant font size. This is a requirement from the perspective of our target applications. Our system does, however, constrain layout dimensions in the horizontal direction, so that only vertical scrolling is needed if the layout can not fit in the desired viewport. In treemaps, siblings are arbitrarily stacked vertically or horizontally at every other step in the recursion, as a way to ensure that both the X and the Y dimension is used. In our own system, tuples are always stacked vertically. Fields are stacked either horizontally, when contained in a nested table, or vertically, when part of an outline view. This ensures one kind of visual consistency while still allowing the widths of layouts to be constrained.

FISH [77] is a variation of the Treemap concept. In this system, styling attributes can be used to configure various node presentation details, including the choice to stack siblings either vertically or horizontally. Such styling must be applied to each individual data node, unlike in our own system, where styling is applied to schema nodes only. Strip Treemap [13] optimizes the presentation of a treemap by sizing rectangles such that they can be stacked in contiguous "strips" without broken horizontal lines. Like Treemap, and

unlike our own system, both FISH and Strip Treemap assume that all data must fit into a viewport of predetermined size both in the horizontal and vertical direction.

Elastic Hierarchies [109] and EncCon [79] both combine node-link representations with some other representation in order to visualize tree-structured data. Node-link representations are typically less space-efficient than nested or indented representations, and seldom appear in traditional database GUIs, our target application. Thus, we did not use them ourselves. EncCon uses node-link representations in every level of the tree, but maintains an invisible Treemap-like layout to determine a favorable positioning for each node. In Elastic Hierarchies, subsequent levels may use either Treemaps or node-link representations, determined interactively. The Elastic Hierarchies paper also has a relevant discussion of the design space of hybrid tree layouts, but does not consider nested tabular layouts. Neither does Graham and Kennedy's survey. This is likely because tabular layouts apply only to structured data, where series of children in a tree all can be expected to have the same substructure. Systems that solve the more general problem of dealing with semi-structured (schema-free) data will not naturally be able to take advantage of tabular layouts.

Nested table layouts are used in Related Worksheets [10] and in applications produced with AppForge [108] or App2You [66]. These systems do not allow the nested table layouts to be re-styled as indented lists or deferred to lower layers of the tree-structure. They can not automatically constrain the width of the layout to the available page width.

VisualXML and XMLAD [30], like Elastic Hierarchies and EncCon, use node-link representations as part of their interface, but are geared towards XML visualization. The XMLAD system does indeed use tabular representations as part of the output layout, but does not support nested tables. Thus, only the bottom logical relations in the data may be displayed as tables. Similarly, the VisualXML system uses outline or list-style representations for the bottom relation level of each subtree. Again, both XMLAD and VisualXML use a node-link representation for all higher levels. This is in contrast with our own system, which automatically uses a column-enabled, indented list for a variable number of top levels in the tree, followed by nested tables for a variable number of bottom levels.

Tree Rewriting provides a visual language semantically analogous to the lambda calculus [58]. While general enough to produce just about any layout from a set of input data, a user must manually specify how these layouts are to be constructed. Unlike our system, Tree Rewriting is unable to produce a default layout subject to a constraint such as available page width. Furthermore, no mechanism is available to help ensure the careful alignment of column fields that is required in order to produce tabular or nested tabular sublayouts.

A relevant class of commercial systems consists of *report generator* tools such as Crystal Reports[1] and Altova XML-Spy/StyleVision[2]. An extensive survey is provided by Król [67]. These systems let the user build output layouts analogous to those produced by our own system, using a variety of input data sources. The layout building process is manual. Altova's Grid View uses a less compact variation of a nested table layout, with table headers repeated for each nested relation value. Navigation requires heavy use of both vertical and horizontal scrollbars as well as manual collapsing and expanding of data nodes. Altova does combine vertical arrangements of tuple fields in an outline-style

[1] http://www.crystalreports.com
[2] http://www.altova.com/xml-editor

sub-view with the use of nested tables, but is unable to render subrelations as indented lists.

2.2.2 Visualization of Flat Tabular Data

We here include systems that visualize flat tables of data, that is, primitive values organized in rows and columns. This excludes systems that can handle tree-structured data, which were discussed previously.

FOCUS [96], TableLens [88], and the system by Tajima and Ohnishi [102] deal with the problem of displaying large, mostly flat tables. The latter system includes a "Record" viewing mode that resembles a single-level form view. While all of these systems support certain cases involving values spanning multiple cells, they do not operate on structured nested data in general. Chi's visualization spreadsheets [28] combine a table layout at the outer level with cells containing plotted 3D graphics, but require the user to define each visualization using commands, and are primarily focused on numerical data.

Show Me [73] is an autostyling system for Tableau, an interactive visualization system. Since Tableau operates on tabular input data and produces pivot table or *crosstab*-based outputs, it does not fall into the tree-structured data visualization category.

2.2.3 Document Layout Systems

Document layout systems, like the one described by Jacobs et al. [52], deal with the problem of rendering a given amount of text with a given font size on a set of dimensionally constrained pages. They are otherwise different from our own system, since they operate on a very different class of input data.

2.2.4 Automatic Form Generation

This category of systems focuses on the generation of form-based user interfaces, like those found in traditional CRUD database applications.

Supple [41] generates widget-based user interfaces for devices of various sizes and contexts using a cost optimization algorithm, but does not deal with table layouts or other layouts that take advantage of repeated structure in data that adheres to a schema. The Right/Bottom strategy [16] defines heuristics for widget placement in dialog boxes, but similarly does not deal with structured relational data.

Database application builder tools such as FileMaker[3] and Microsoft Access[4] include wizards to help with the creation of new forms, but are only able to layout out a single level of fields automatically and, unlike our own layout system, can not use statistics about the data in each field to make structural layout decisions.

[3] http://www.filemaker.com
[4] http://office.microsoft.com/access

Chapter 3

A Visual Query Language

3.1 Introduction

Four decades after Query-by-Example, 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. We agree with Liu and Jagadish [71] that a successful alternative 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 progressively refined visual query [71]. 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. [71]

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 chapter, 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

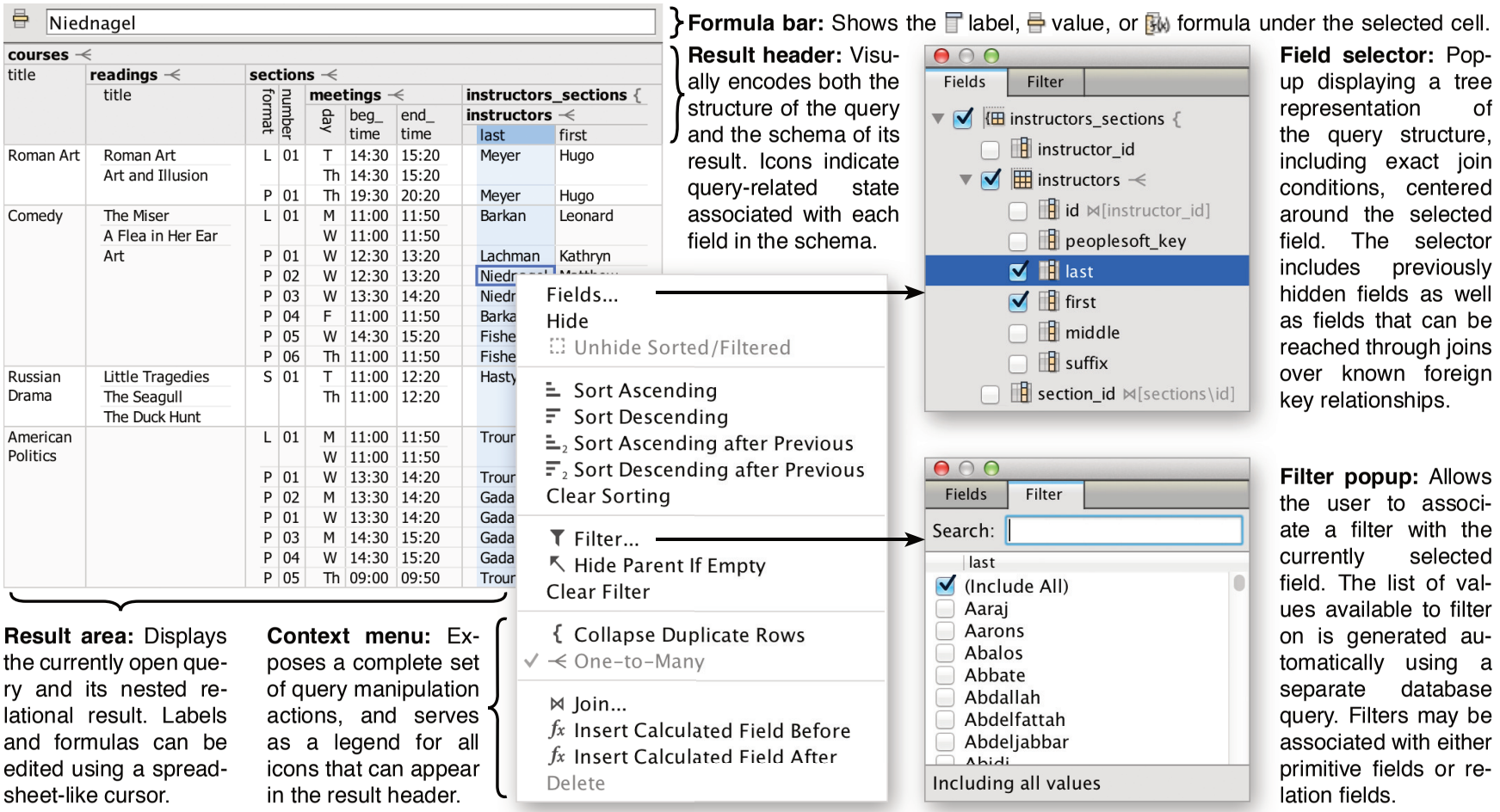


Figure 3-1: The SIEUFERD query interface. To create queries, users start from a simple tabular view of a table in the database and add filters, formulas, and nested relations. The integrated result and query representation is displayed continuously as the user interacts with the data. The particular 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.

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 [53, 68], and display results using a nested table layout (see Chapter 5).

The user interface of our visual query system is shown in Figure 3-1. At any given time, the user will be looking at the nested relational result of the visual query currently being built, formatted using a nested table layout. All query manipulation actions are initiated from the result layout, satisfying requirement R1.

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 nested data model. 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. The user can then manipulate any part of the query by initiating an action on the corresponding field in the result layout, satisfying requirement R2. 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.

Using spreadsheet-like constructs such as formulas and filters, the user can express a relationally complete [32] set of query operators plus calculation, aggregation, outer joins, sorting, and nesting (see Section 4.6 for details). This covers the full set of query operators generally considered as the minimum to model SQL [12, 49], and expresses, for example, all `SELECT` statements valid in SQL-92[1]. This satisfies requirement R3.

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. 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) [17] 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.

[1]To empirically verify the list of relational operators required to express SQL-92, the author personally contacted two implementors of SQL-92 compliant database systems, Julian Hyde of the Apache Calcite project and Thomas Neumann of the HyPer project, who helpfully, and independently, listed the minimum set of (bag-based) relational operators required. (Thank you!) Neumann notes that more operators may be required to implement an efficient query processor. The latter is not of concern for the discussion of our visual language's expressiveness.

3.2 System Description

3.2.1 Overview

Our core query building interface was shown in Figure 3-1. 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 (columns) are nested under others in the nested data model. The term *schema* here refers to the definition of fields and sub-fields in the query's result rather than the schema of the underlying database. 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. When nested relational data is shown using a nested table layout, the terms *field* and *column* are equivalent. Since our system can also display data using various non-tabular layouts (see Chapter 5), we use the more generally applicable term *field*.

Starting from any selection of fields 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. 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, and to display exact join conditions.

3.2.2 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, such as the state of a filter or the contents of a formula, on each field. We refer to

the annotated schema as the SIEUFERD *query model*. The term *model* is used here as in the "model-view-controller" architecture; it denotes the portion of our system that maintains the underlying state of the user interface. 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. In this chapter, we will explain the SIEUFERD query model entirely in terms of its visual representation. Its internal representation, expressiveness, and translation to SQL will be discussed in Chapter 4.

Table instantiation. 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

Conceptually, the gray header area represents the query, while the area below shows the result of the query. The header shows the *relation* field COURSES in bold font with its *primitive* child fields (ID, AREA_ID, etc.) below. Each row in the result is a *tuple*. In our query model, 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.

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 3-1). The one-to-many icon (≺) on the READINGS relation indicates that our system determined the latter may contain more than one tuple for each corresponding tuple in COURSES, the parent relation. The icon is also displayed on the COURSES, indicating (as is common) that the root relation may contain multiple tuples. Our system automatically makes these determinations based on primary key constraints declared in the database and the join constraints set in the user's query.

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* [97], 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. This behavior can be changed by means of the HIDE PARENT IF EMPTY section, discussed later.

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 semantics allows us to hide several fields from the screen without altering the meaning of the joins or disturbing the order of the tuples:

courses ≺					
title	may_ pdf	may_ audit	exam_ type	readings ≺	
				author_name	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 (COURSES, AREA, READINGS, SECTIONS, MEETINGS) 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					
Social Analysis	SA	American Politics			L	01	M	11:00	11:50
							W	11:00	11:50
					P	01	W	13:30	14:20
					P	02	M	13:30	14:20
					P	01	W	13:30	14:20
					P	03	M	14:30	15:20
					P	04	W	14:30	15:20
					P	05	Th	09:00	09:50
		Junior Seminars	Pierre Loti	India	S	01	M	13:30	16:20
Social	SA	Judicial Politics	Rosenberg	The Hollow	L	01	W	11:00	11:50

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 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. The latter is a fact that the user cannot deduce by looking at the query result alone, since the absence of a one-to-many relationship in the currently visible excerpt of the data is no guarantee that such relationships may not exist elsewhere in the dataset, or when the query is rerun on a future version of the dataset. Having a good mental model of where a one-to-many relationship might occur is important when the user is constructing calculations, as the presence of a one-to-many relationship typically implies the need to use aggregate function to summarize data (see below).

A query may instantiate the same table more than once, from different relations. A relation is thus analogous to a `FROM` clause term in a SQL query. The immediate child fields of a relation always include a set of primitive fields each associated with a column in the instantiated database table. Using the field selector (see Figure 3-1), the user may hide or show these primitive child fields to achieve the effect of including or excluding fields from the query's SELECT clause.

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 AUTHOR_NAME, then by TITLE:

courses								
title	max _enroll	readings		sections				
		author_ name	title	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

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. It is possible to sort on both primitive and relation fields. Sorting on a relation field causes the sort terms of said relation to be included in the sort terms of the parent relation. In the following example, following the rules above, the COURSES relation is effectively sorted first on AREA\CODE, then on AREA\ID, then COURSES\TITLE, then COURSES\ID:

courses		area					
id	area_id	⋈ id	title	code	title	may_pdf	may_audit
177	2	2	Literature and the Arts	LA	Comedy	Y	Y
56	2	2	Literature and the Arts	LA	Roman Art	N	Y
845	2	2	Literature and the Arts	LA	Russian Drama	N	N
1795	4	4	Social Analysis	SA	American Politics	Y	Y
3921	4	4	Social Analysis	SA	Judicial Politics	Y	Y
2566					Junior Seminars	N	N

Sorting on a relation makes the most sense when the relation in question (e.g. AREA) is known to have at most one tuple for each corresponding tuple in its parent relation (COURSES). Fields used for sorting (e.g. AREA\CODE) are projected into the sorted relation (COURSES) by the same mechanism that handles non-aggregate inward references in formulas; this handles the one-to-many case as well. See *flat joins* below.

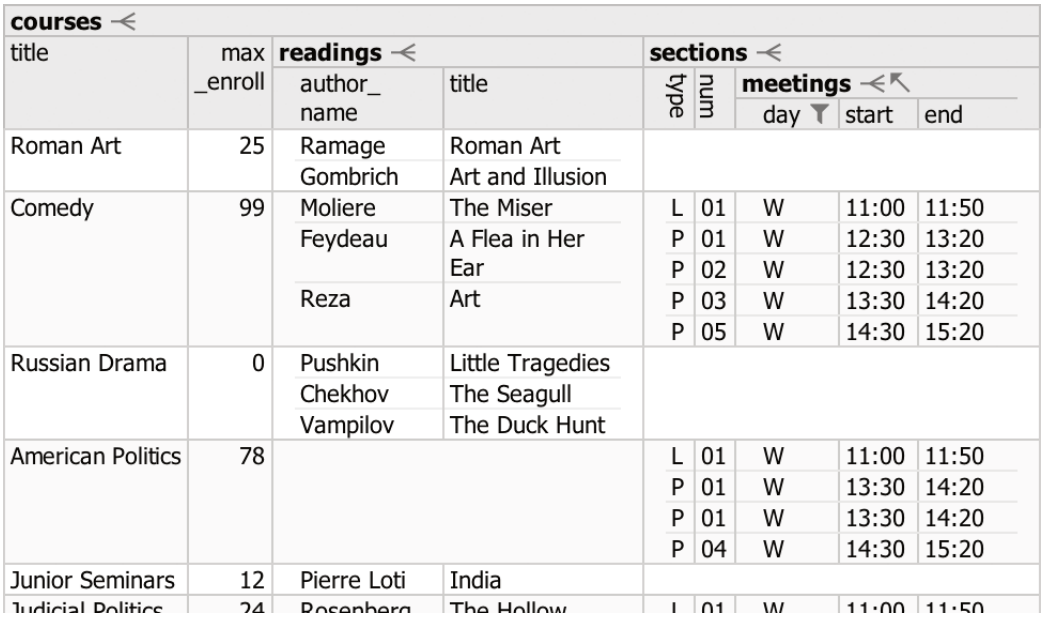
Filter. Using the filter popup (Figure 3-1), a filter can be defined on any field, indicated by the filter icon (▼). Filters on primitive fields restrict the tuples of their containing relation, while filters on relation fields restrict the set of tuples retrieved in that relation. In the following example, the MEETINGS relation is filtered to show only tuples for which the DAY is W:

courses		readings		sections		meetings		
title	max_enroll	author_name	title	type	num	day ▼	start	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

In our current implementation, filters can be defined to either select a set of values to include (e.g. "only values M,W,F"), or to select a set of values to exclude ("all values except M,W,F"). Other kinds of restrictions, such as ranges ("values between 5 and 10") or string matching ("values that start with INTRO"), can be expressed as filters on formulas (see below), but could also be integrated directly into the filter state for improved usability.

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 (↖), as seen on the SECTIONS and MEETINGS relations in the previous example. 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		sections				
		author_ name	title	type	num	meetings day	start	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, filtered, or joined on.

SIEUFERD formulas are syntactically similar to spreadsheet formulas, allowing literals, arithmetic operators, and named functions, 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 so 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 [55, 23], 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. Our system's formula language does not provide a relational algebra, but rather allows simple computations over primitive values.

We will elaborate a bit on the previous point. Consider, hypothetically, that we decided to design our system such that formulas could return entire relations rather than only primitive values. To accommodate this, the formula language would have had to include not only scalar and aggregate functions, but also functions returning relations and performing operations on them. We already have a language that defines such functions—namely the relational algebra, and its equivalent, SQL. Since the goal of our system is to provide a vi-

sual alternative to SQL, we consider it unacceptable to embed a SQL-like language within our system's user interface.

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

courses

title	sections						
	type	num	meetings				
			day	start	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])

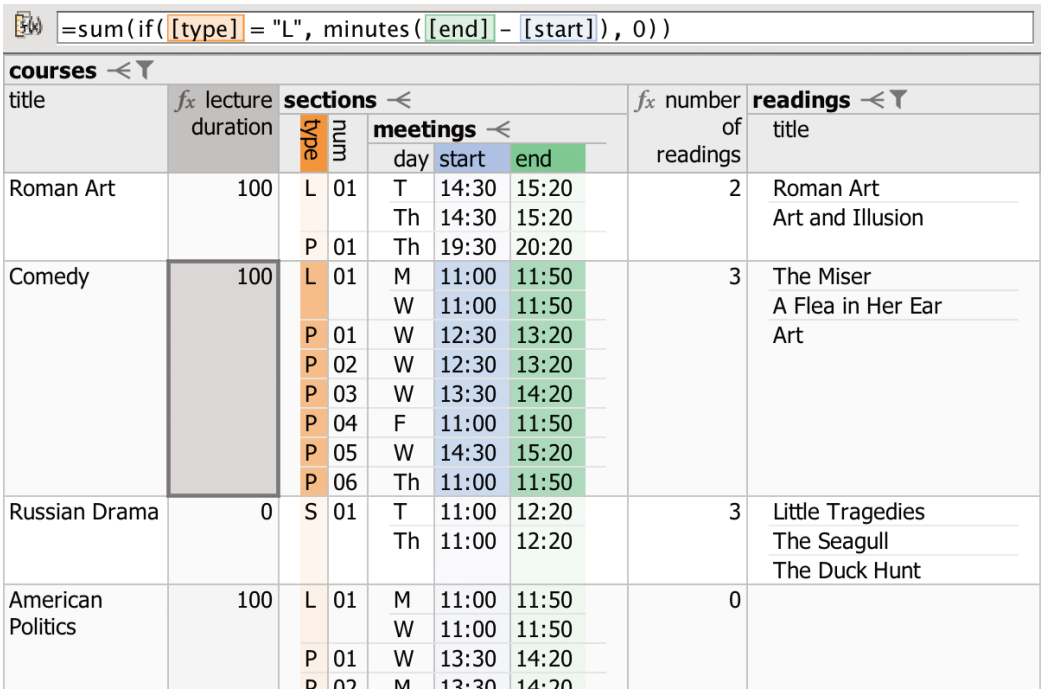
courses

title	f_x total duration	sections					
		type	num	meetings			
				day	start	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.

Aggregate functions, with the exception of MIN and MAX, are sensitive to duplicates in input values, as illustrated by the correct totals over identical DURATIONS in the example above. The bag-like semantics of aggregate functions can be understood by considering a formula like =SUM([MEETINGS\DURATION]) to be shorthand for a more explicit notation $=\text{SUM}_{field1}$([MEETINGS\DURATION], [MEETINGS\ID]), where the aggregate function SUM_{field1} constructs a set of (DURATION, ID) tuples and then calculates the sum from the value in the first field of each such tuple. Whenever a primitive field (e.g. DURATION) is referenced from within the argument of an aggregate function, a tuple-identifying subset of fields from the referenced field's containing relation (e.g. MEETINGS) is automatically included in the aggregate input (e.g. the hidden primary key field MEETINGS\ID in the example above). Note that our backslash notation serves only to make it clear which relation each referenced field belongs to, for cases where there are multiple fields with the same label (e.g. ID).

The following query will illustrate a few more ways in which aggregate functions can be used and combined:



courses

title	fx lecture duration	sections: type	sections: num	meetings: day	meetings: start	meetings: end	fx number of readings	readings: title
Roman Art	100	L	01	T	14:30	15:20	2	Roman Art
				Th	14:30	15:20		Art and Illusion
		P	01	Th	19:30	20:20		
Comedy	100	L	01	M	11:00	11:50	3	The Miser
				W	11:00	11:50		A Flea in Her Ear
		P	01	W	12:30	13:20		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		
Russian Drama	0	S	01	T	11:00	12:20	3	Little Tragedies
				Th	11:00	12:20		The Seagull
								The Duck Hunt
American Politics	100	L	01	M	11:00	11:50	0	
				W	11:00	11:50		
		P	01	W	13:30	14:20		
		P	02	M	13:30	14:20		

The highlighted formula, using again the more explicit notation from the previous paragraph, would be evaluated as $=\text{SUM}_{\text{IF}(field1=\text{"L"},\text{MINUTES}(field2-field3),0)}$([SECTIONS\TYPE], [MEETINGS\END], [MEETINGS\START], [SECTIONS\ID], [MEETINGS\ID]), first constructing a set of (TYPE, END, START, SECTIONS\ID, MEETINGS\ID) tuples and then computing the sum of the values yielded by evaluating the argument expression for each input tuple. Another formula in the example above is that of the NUMBER OF READINGS field; it uses the COUNT aggregate function to count the number of tuples in the READINGS relation. The COUNT aggregate is the only function in our formula language that permits a relation field as an argument; the NUMBER OF READINGS formula could be written as either =COUNT([READINGS]) or, equivalently, =COUNT([READINGS\ID]) or =COUNT([READINGS\TITLE]). Any non-nullable field in the READINGS relation could be used in the latter case. Finally, note that the example query above

computes two independent aggregates (LECTURE DURATION and NUMBER OF READINGS), each over its own input relation (SECTIONS and READINGS, respectively). This is one example of a *multi-block* query, that is, a query which requires multiple `SELECT` clauses in the corresponding generated SQL query. Another example of a multi-block query would be a query that uses the output of one aggregate function as an input to another. Such queries are fully supported by our system.

Filters and aggregate functions. When an aggregate function references a relation with a filter applied to it, the filter is evaluated before the aggregate. This allows conditional sums to be computed without the use of =SUM(IF(...))-type formulas, while always seeing the exact tuples that contribute to the total. In the following example, we filter the SECTIONS relation to only include lecture-type sections. There are also some empty sets of sections. The TOTAL DURATION for each course changes accordingly:

courses								
title	fx total duration	sections						
		type	num	meetings				
				day	start	end	fx duration	fx 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 field containing the aggregate function, e.g. TOTAL DURATION in the example above, or its sibling fields, e.g. TITLE. Such filters are analogous to `HAVING` and `WHERE` clauses, respectively, in a SQL query block.

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			readings	
title	exam_ type	author reference f_x	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 (e.g. AMERICAN POLITICS from COURSES). Furthermore, a scalar formula like =IF(FALSE, [INREF], 42) can safely be constant-folded or otherwise considered equivalent to =42. This would not be the case if formula references were resolved using inner joins rather than left joins. Finally, formulas may use null values generated by left joins to detect missing values. In the following example, a presentation title is generated for each course, consisting of the AREA of each course followed by its TITLE. Courses with no assigned AREA (JUNIOR SEMINARS) are handled as a separate case:

=if([area\title] is null, [title], concat([area\title], ": ", [title]))

courses						
prettified title f_x	area		title	may_ pdf	may_ audit	exam_ type
	title	code				
Literature and the Arts: Roman Art	Literature and the Arts	LA	Roman Art	N	Y	Other
Literature and the Arts: Comedy	Literature and the Arts	LA	Comedy	Y	Y	Final
Literature and the Arts: Russian Drama	Literature and the Arts	LA	Russian Drama	N	N	Other
Social Analysis: American Politics	Social Analysis	SA	American Politics	Y	Y	Final
Junior Seminars			Junior Seminars	N	N	Other
Social Analysis: Judicial Politics	Social Analysis	SA	Judicial Politics	Y	Y	Final

The main use case for non-aggregate inward references is when the referenced relation (e.g. AREA) is known to have at most one tuple for each corresponding tuple in the referencing relation (COURSES). This may be a guaranteed consequence of the child relation being joined on its instantiated table's primary key, as when traversing a foreign key relationship in the forward direction, or simply a property of the input data. If neither is the case, a user who entered a non-aggregate inward reference may be surprised to see duplicated tuples in the referencing relation, or miss this nuance of formula semantics completely. This is a potential usability problem. In the future, we may require inward references that pass

through one-to-many relationships (≺) to always be enclosed in either an aggregate function or an explicit UNNEST function, otherwise showing the user a warning message (e.g. "Reference to READINGS may yield more than one value; must be enclosed in an aggregate or UNNEST function."). The UNNEST function would serve to suppress the warning and enable the current behavior. An analogous UNNEST function exists in PostgreSQL, where it is used to unpack and join in tuples from array values.[2]

Set projection. By default, tuples internally 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 on the screen. 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 COLLAPSE 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

courses ≺	
title	sections ≺ {
	type
Roman Art	L
	P
Comedy	L
	P
Russian Drama	S
American Politics	L
	P
Junior Seminars	S
Judicial Politics	L
	P

When a relation has the COLLAPSE DUPLICATE ROWS option enabled, hiding a primitive child field may cause the number of tuples returned to decrease, as only visibly unique tuples are retrieved. This can be seen in the transition from the second to the third panel above, as the STATUS field is hidden.

When an aggregate function references a relation that has the COLLAPSE DUPLICATE ROWS option enabled on it, the set projection on the latter relation is evaluated before the aggregate, as for filters. Thus, as before, the tuples that are used as inputs to aggregate functions are the same tuples that the user can see on the screen:

[2] http://www.postgresql.org/docs/9.6/static/functions-array.html

=count([sections])

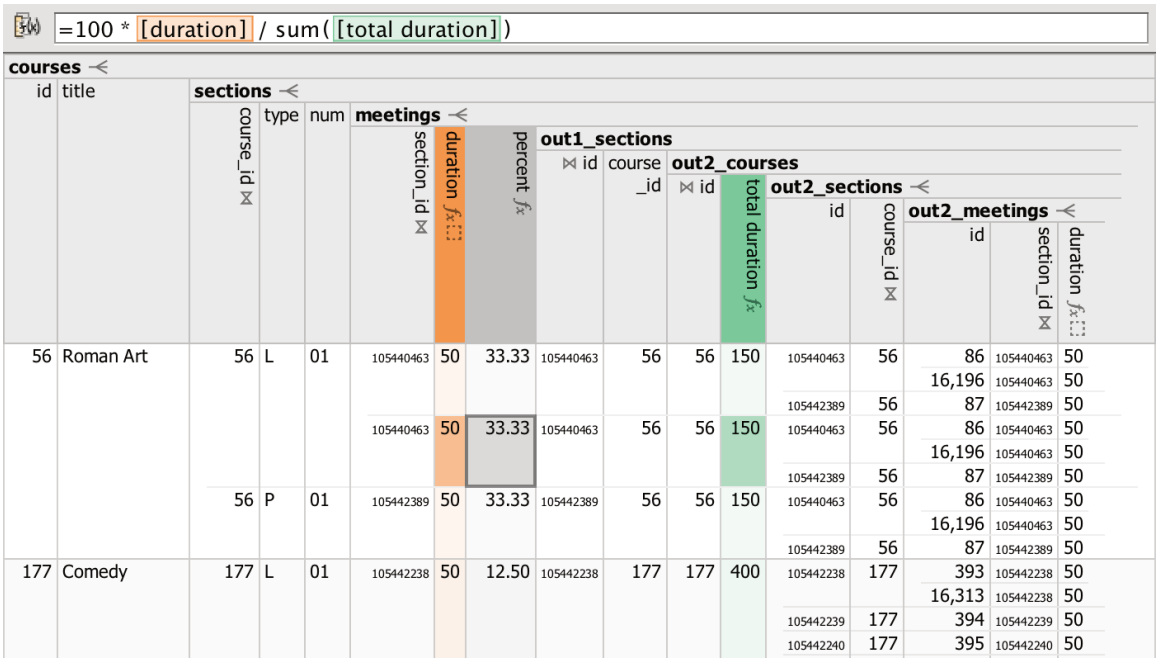
courses ≺			
title	f_x count	sections ≺{	
		type	status
Roman Art	2	L	O
		P	X
Comedy	3	L	O
		P	O
		P	X
Russian Drama	1	S	O
American Politics	2	L	O

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

=100 * [duration] / [total duration]

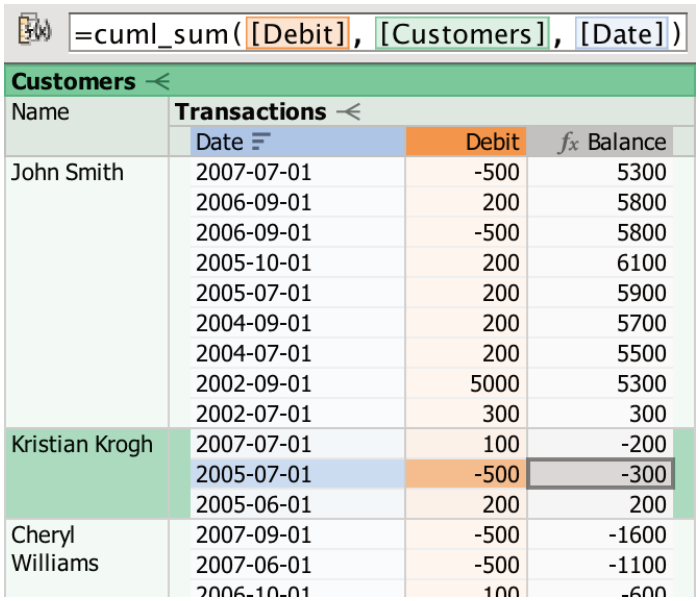
courses ≺								
title	f_x total duration	sections ≺						
		type	num	meetings ≺				
				day	start	end	f_x duration	f_x percent
Roman Art	150	L	01	T	14:30	15:20	50	33.33
				Th	14:30	15:20	50	33.33
		P	01	Th	19:30	20:20	50	33.33
Comedy	400	L	01	M	11:00	11:50	50	12.50
				W	11:00	11:50	50	12.50
		P	01	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 [105, p. 8]. More specifically, we create a copy of the referenced relation as a descendant of the referencing formula's containing relation, and then replace the original outward formula reference with an equivalent inward reference to the descendant copy. The decorrelation is done one relation level at a time, with equijoin constraints being applied to match tuples in the original and copied relations. For the example query above, a decorrelated equivalent would be the following:



More efficient decorrelation methods may be possible in certain cases.

Window functions. While not part of the operators we need for expressiveness at the SQL-92 level, it is also possible to support *window functions*, such as cumulative sum and moving average, from within the visual query interface. Such functions were introduced in SQL:2003, and are supported for instance by PostgreSQL and Oracle, but not by MySQL or SQLite. As an example, the formula below is translated to the SQL clause `SUM(Debit) OVER (PARTITION BY Customers.ID ORDER BY Date)`:



In the example above, a list of (DATE, DEBIT) transactions are added up in order of increasing dates to yield the balance after each day. The first argument to the CUML_SUM function is the primitive value to add at each iteration. The second argument is a relation

reference that specifies the level at which data should be partitioned; the key fields of that relation are used as the partition key in the `PARTITION BY` clause. The third argument is the field to order on for the purposes of evaluating the window function. While in the example above, the window function happens to be ordered on the same field (DATE) on which the result of the TRANSACTIONS relation is ordered, this need not be the case. For instance, the user could now choose to sort results descending on the BALANCE column to show the days with the highest balance first.

Note that when two transactions occur on the same date (e.g. 2006-09-01), the balance at the *end* of the day is shown for both transactions. This follows the semantics of SQL's `PARTITION BY` clause, which ensure that the result of window functions are deterministic with respect to partial orders.

Allowing ordering to be specified as an argument to a formula function like CUML_SUM is a bit of a kludge, since our system already has a different, more visual interface for specifying the order of query results, and since desired orderings may be more complex than a simple ascending sort on a single field. It is at the same time undesirable to make the overall visual query system more complicated in order to make a presumably infrequently used feature easier to use. A compromise could be to allow the user to omit the ordering clause when first writing a window function formula, instead relying on the existing result ordering, but then automatically rewriting the formula to insert the relevant order definition if the user reorders the formula's containing relation.

The example above was for a cumulative sum; the implementation and design considerations would be similar for other window functions.

A particular use case for window functions is to be able to express complex top-K queries such as "average sales by state of top 10 dealerships in each state". In the latter example, the input to the aggregate function would be filtered to include only results for which the ROW_NUMBER window function is less than or equal to 10. In PostgreSQL, such queries may be expressed either using the ROW_NUMBER window function or a `LIMIT` clause in a correlated subquery, but SQL-92 includes neither, and cannot express such queries.

3.2.3 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:

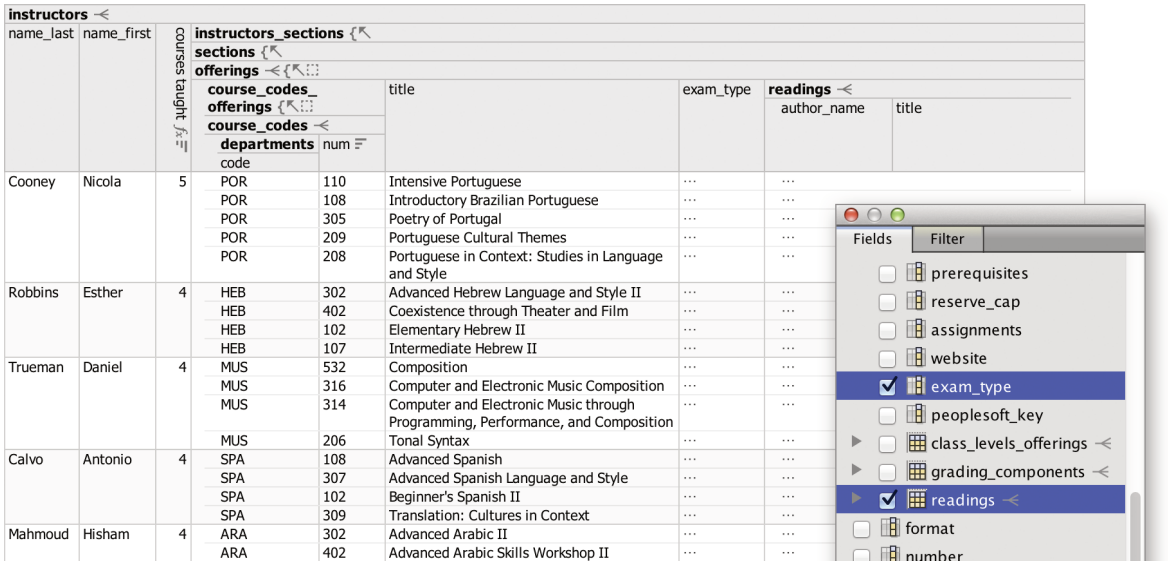


Figure 3-2: Temporary layout displayed during execution of a long-running (~1900ms) query. The user has just unhidden the EXAM_TYPE and READINGS fields. The unhidden fields are immediately displayed using placeholder icons (⋯); meanwhile, generated SQL queries run in the background to retrieve an updated result for the entire visual query.

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. See Section 5.4.2. 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. Such visual stability is important for usability, because it allows the user to easily spot which parts of a result actually changes in response to a modification to the query.

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. Such fields can still be displayed at the correct width as long as the system has had a chance to measure values for that field at some earlier point in the query building session (see Section 5.4.2). 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. See Figure 3-2.

For example: When unhiding a previously hidden field, the user sees the result layout

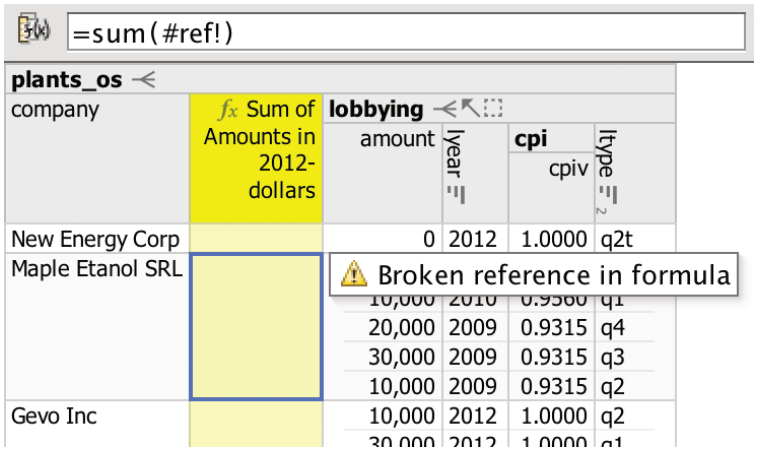


Figure 3-3: High-level error handling. A referenced field was deleted, so the formula can no longer be evaluated. The system shows a warning while evaluating the rest of the query normally.

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. When hiding a field, the temporary layout usually ends up being identical to the final one. Pure presentation changes, such as the user editing a field label or the system automatically remeasuring column widths (see Section 5.4.2), do not cause queries to be re-executed.

Interruptable queries. If the user modifies the query before the previous query has finished executing, the previous query is automatically interrupted using the database back-end's preferred mechanism. This is crucial for letting the user escape from long-running queries, and also allows the user to perform multiple modifications 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. For instance, a user might speed up a query like "find the companies with the greatest total reported lobbying expenditures" by temporarily filtering to only include lobbying reports for a certain year. The user can then continue to tweak the way totals are calculated, doing complex calculations to correct for inflation and so on, with intermediate results being shown quickly at all times, and then only finally clear the original filter to run the slower query across all lobbying years. This workflow is also a good example of why it is important to let users easily modify operations, such as filters, that were specified many steps earlier.

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.

Table 3.1: 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.

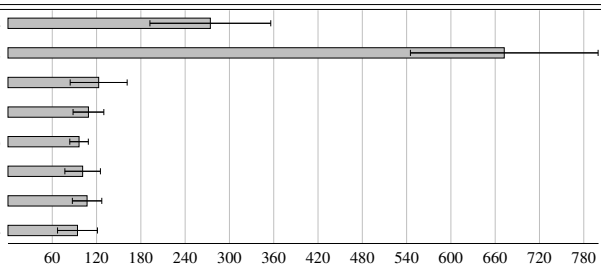
High-level error handling. User-defined formulas introduce a variety of possible error conditions, including circular references, broken references, type errors, and arithmetic runtime errors. Our system detects and handles many such errors before they can reach the database layer. This lets us produce more user-friendly error messages than if we had tried to execute an erroneous generated SQL query and then passed the resulting error message from the database and back to the user. 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 compiled to literal null values, ensuring that the rest of the query can still be evaluated normally. See Figure 3-3.

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 is supported by storing successive states of the modified query model; a similar technique is used in Tableau [100, p. 90]. Like other kinds of query modifications, undo/redo benefits from several of the previously mentioned features, e.g. interruptable queries.

Table 3.2: 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	Manual join	2a	7	274
1b	Totals	Formula	2b	7	672
2a	Inflation	Manual join		7	123
2b	Correction	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.

3.3 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 3.1 for a demographic summary. Most of the participants use Excel daily, or have had some need to work with structured data in their jobs.

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 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.

3.3.1 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 3.2.

Formulas and manual joins. Tasks 1 and 2 correspond to the lobbying example from Section 1.1, 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 has 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 on the formula portion, 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 3-1, 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 about 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. Our timing data shows no significant difference between training and follow-up tasks involving auto joins.

3.3.2 Observations

We now discuss a specific observations gathered from both the standardized and the open-ended portions of the study. A selection of observations is summarized in Table 3.3.

Mode of interaction. A recurring theme in users' initial attempts at using our system was to explore areas outside the direct manipulation area: toolbars, menus, and a list of database tables shown in a sidebar. Users CDFGH assumed that joins would need to be initiated from somewhere outside the result area, such as by drag-and-drop in the list of database tables, or by dragging a table into the result area. Such drag-and-drop interactions are common in tools like Tableau and Microsoft Access. We gave users the hint that all

Table 3.3: Selected observations from the formative user study.

Title	*Observation ("The user...")*	*N*	*Users*
Query construction; fields and joins			
Pulling data into view	First assumed that joins would need to be initiated from outside the direct manipulation area.	5	CD FGH
Tried to drag tables	Attempted drag-and-drop operations in the table list outside the direct manipulation area.	3	F H M
Popular auto joins	Reacted enthusiastically to auto join behavior (e.g. "fantastic", "wow", "damn", "amazing").	6	E GH JKL
Query construction; formulas and aggregates			
Sum action	Before learning they would have to write a formula, looked for an explicit "sum" action.	4	BCDE
Formula builder	Looked for an Excel-style formula builder.	3	C G K
Sum without formula	Suggested having a method to do a sum without manually having to write a formula.	2	A M
Situational awareness			
Verifying aggregates	Tried to visually verify the output of aggregate functions.	2	AB
Verifying joins	After first manual join, wanted to visually verify that join condition's two sides were equal.	3	C E J
Bad default fields	Had a harder time quickly grasping results due to bad field visibility defaults after joins.	8	ABCDEFG I
Massive lists of fields	Had a harder time navigating schema in field selector due to large number of primitive fields.	5	E IJ LM
Schema diagram	Asked for or suggested a schema diagram feature (besides diagrams handed out on paper).	3	F J M
Understanding the query model; fields and joins			
Identified need for join	In task 2a, quickly understood that another join was required, and performed it correctly.	5	ABC EF
Needed hint to use join	In task 2a, after the learning task, required a hint that another join was required.	2	D G
Hidden ancestors	In field selector, was briefly confused by checked descendants of expanded unchecked fields.	4	B FG I
Hiding relations	When asked to hide a relation, selected all child fields and invoked HIDE.	3	BC F
Understanding the query model; formulas and aggregates			
Where to put formula	After inserting a calculated field, had initial trouble learning where to enter the formula.	4	DEF K
Formulas made sense	Noted that the all-column behavior of formulas and formula references made sense.	2	I L
Aggregates made sense	Noted that aggregate function behavior, incl. grouping and subtotaling behavior, made sense.	4	C E G K
Tried column reference	Correctly assumed aggregate argument would be a column reference, not a range of cells.	5	ABC F H
Tried range reference	First time, incorrectly assumed aggregate argument would be a range of cells, as in Excel.	4	DE G N
Formula at wrong level	First time, incorrectly placed aggregate formula in same relation as argument to aggregate.	4	C FGH
Understanding the query model; filters			
Filters and joins	Had no trouble understanding how filters worked, including interaction with multi-level joins.	4	B F H K
Filters and aggregates	Had no trouble understanding interaction between filters and aggregate functions.	4	C F H K
Root sort to search	Sorted a large table (728K rows), then scrolled downwards trying to search for items visually.	2	A G
Deep sort to search	Sorted on field in deeply nested relation, hoping for root relation to be sorted on said field.	3	G L N
Infrequent filter user	Reported not being a frequent user of Excel's "filter" feature.	3	A G N
Analogies to other tools			
Thought of pivot tables	Mentioned thinking about pivot tables/other crosstab interfaces when first attempting tasks.	5	BC E HI
Intuitive selection/menu	Had an easy time performing Excel-like column selection and operating the context menu.	6	B DE GHI
Looked for "unhide"	Looked for a context menu action specifically named "unhide", as in Excel.	3	I K N
Field selector parent	In field selector, looked for "parent directory" button like that of Windows XP's file picker.	2	L N
Other details			
Join dialog easy	When doing first manual join, had no trouble with the JOIN dialog.	6	ABC E GH
Join dialog one stumble	When doing first manual join, stumbled once in the JOIN dialog, but quickly recovered.	2	D F
Drag in join dialog	Tried drag-and-drop in the join dialog. Not necessary, but did not cause trouble either.	3	ABC
Distracting "group"	Was confused by a non-essential shortcut action called "group", which was later removed.	4	BC F H
What to COUNT	Used the COUNT aggregate with a primitive argument, e.g. =COUNT([TITLE]).	3	B HI

actions required for tasks would be found in the central direct manipulation area and its context menu. Users had no trouble operating the Excel-like selection and context menu.

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 COLLAPSE 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 while struggling with the latter problem:

"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."

User C: *"Hmm, you can't do calculated field by row, it only does it by column."* Asked where the user would want the sum to appear: *"Either in the top row, next to each name, or at the bottom of each section."*

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. Small improvements could also be made to make it easier to discover how to edit formulas.

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."

Users BHI, when asked to do a counting task, such as "find the instructor teaching the highest number of courses", used the COUNT aggregate function with a primitive field as an argument, using either the technical primary key (e.g. ID) or another field that the user presumed to be an identifier for tuples in the relation to be counted (e.g. TITLE). User H noted that "this might not be the ideal way to do it" due to the potential for duplicated titles. This suggests that users suspected that the COUNT aggregate would collapse duplicate input values, which is not actually the case. User B, when asked how one might count the actual rows in the relation, inserted a constant formula with the numerical value "1" and used a SUM function to tally up the constants. None of the users discovered that a relation reference can be used in the COUNT aggregate instead of using an arbitrary primitive field from the same relation.

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", as 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."

User J: *"It was so easy. It was extremely easy to do this. The idea of just being able to just simply right-clicking on a column and identify what I want to include in this thing without having to think about a SQL or Access query, it was just very intuitive."*

User L: *"It's pretty fast! ... For what it's doing, it's amazing."*

Field selection; conceptual model. One source of confusion was the fact that the field selector may show a field as checked even when an ancestor relation is not. Such states are permitted because they may be semantically significant in certain cases (e.g. when a hidden relation is used as an input to an aggregate function and the COLLAPSE DUPLICATE Rows option is enabled on said relation). Users BFG expanded an unchecked relation field in the field selector and were surprised to see child fields already checked despite not yet being visible in the result area. This was due to our system attempting to set defaults for visible fields in all newly introduced relations, hidden or not. User I, in contrast, attempted to uncheck a parent relation after first checking several child fields that were meant to be included in the result. This had the effect of hiding from view even the desired child fields from the result. While users quickly recovered from these problems, the field selector could be improved by using an alternative checkmark indication for fields that descend from hidden relations.

Users BCF, when asked to hide a relation, multiple-selected all of its child fields in the result area and invoked HIDE on them. We later modified the HIDE action to recognize this pattern and hide the common parent relation instead of each of its individual child fields.

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, we configure this setting automatically. Various heuristics could work, including attribute ranking algorithms [34, 76], but for now, we simply look for column names containing the words "title" or "name".

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. Alphabetization would be problematic, as users often expect fields to be in a certain order (the phone number always precedes fax number). A search box in the field selector, on the other hand, like that of the filter popup, could work, 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.

Filters. In task 5 and elsewhere, most users had no trouble applying the filter feature. The exception was users AGLN, who first approached filter tasks by attempting to sort rather than filter. Three of these users, users AGN, reported not being frequent users of filters in Excel. Users had no trouble understanding the interaction between filters and aggregate functions, nor with the behavior of filters on deeply nested relations. User E solved a conditional sum task using a =SUM(IF([CONDITION],[AMOUNT],0)) formula instead of using the filter feature, but understood the filter approach as well.

3.3.3 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."

User H: *"It feels like the learning curve was very fast. I mean, I felt like I didn't know much to begin with, but then I feel really comfortable with it now. I could totally do things with it, if I had it."*

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."*

User L: *"This is much more sophisticated than Excel. I think, if you know Excel, at the intermediate level or above, then just playing around with this, you can figure it out."*

User E, on how one might use Excel to solve tasks similar to those given in the study: *"I couldn't imagine doing that [course catalog] activity in Excel. The first example was simple enough, but once you start to do a couple of VLOOKUPs, I think you're starting to go beyond what Excel is really about."* (VLOOKUP is a function that can be used in Excel formulas to pull in data from other worksheets or cell ranges, analogous to a join.)

Table 3.4: Tasks used in the controlled study. Some additional bonus tasks were also available to users who finished quickly. The database used is the 7-table version of the "Northwind" example that shipped with older versions of Microsoft Access.

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

3.4 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).

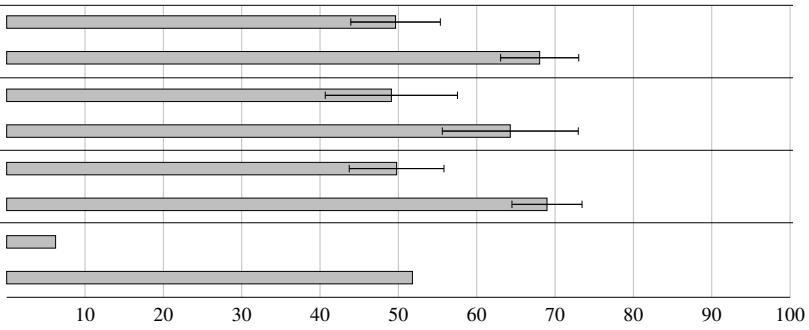
The controlled study was a within-subjects counterbalanced design, measuring usability using the System Usability Scale (SUS) [17]. Tullis and Stetson [103] recommend sample sizes of 12-14 users to get reasonably representative results from within-subjects studies based on the SUS survey; we collected data from 14 users (5 male, median age 36). See Table 3.1 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

Table 3.5: 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



well as multi-block queries (e.g. aggregates used as inputs to other aggregates). The exact tasks are listed in Table 3.4. In both tools, we configured foreign key relationships up front 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 3.5. The raw SUS score is reported along with separate Learnability and Usability scores as defined by Lewis and Sauro [69], as well as a percentile rating among 30 other studies in the B2B (Business Software) category as detailed by Sauro [91]. 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.

3.5 Berlin/BESDUI Benchmark

One limitation of the previous user studies is that the choice of tasks was made by the same researcher who developed the software being evaluated. Another limitation was the

Table 3.6: Summary of BESDUI benchmark results for SIEUFERD, compared with existing results for two other systems. The *capacity* indicates whether or not the query in question can be expressed in each system (as per the benchmark's notation).

	Virtuoso Facets		*Rhizomer*		*SIEUFERD*		
Task #	*Capacity*	*Time (s)*	*Capacity*	*Time (s)*	*Capacity*	*Time (s)*	*Time Relative to Best*
1	100%	27.4	0%		100%	30.7	1.1
2	100%	28.7	100%	12.0	100%	28.9	2.4
3	100%	5.2	100%	5.0	100%	8.3	1.7
4	0%		0%		100%	48.0	
5	0%		0%		100%	55.2	
6	0%		100%	48.4	100%	67.0	1.4
7	0%		100%	3.3	100%	6.7	2.0
8	100%	30.9	100%	20.3	100%	40.0	2.0
9	0%		0%		100%	22.7	
10	100%	2.6	100%	2.6	100%	18.4	7.1
11	100%	38.8	100%	21.4	100%	39.8	1.9
12	50%		0%		50%		

time we had available with each study participant; our hour-long controlled study left only 10-15 minutes to solve unguided tasks on each of the two systems under test, after a similarly short sequence of training tasks on each system. The BESDUI benchmark [42] suggests using a set of 12 queries, sourced from the earlier systems-oriented Berlin SPARQL Benchmark [15], with a keystroke-level model (KLM) [21] to estimate the performance of expert end-users operating a visual query system. Applying the BESDUI benchmark on the SIEUFERD system gives us an opportunity to do an objective comparison of both performance and functionality relative to other visual query systems evaluated using the same benchmark. Furthermore, the detailed enumeration of steps required to construct each benchmark query serves as good examples of the capabilities of the system and how an expert user would make use of them in practice.

The detailed interaction steps required to construct each of the 12 queries in the BESDUI benchmark are listed in Table 3.7. The BESDUI benchmark, like the Berlin SPARQL Benchmark, can be used with both RDF stores and relational databases; for the relational SIEUFERD system we have assumed that foreign key relationships have been declared at the database level as part of the schema definition. To date, the BESDUI benchmark has been applied to two other systems; Rhizomer [18] and Virtuoso Facets[3]. Figure 3.6 shows the results for SIEUFERD compared with Rhizomer and Virtuoso[4]. Task 12 consists of a query part and a data export part; SIEUFERD can do the former but not the latter, Rhizomer vice versa.

Comparing the results in Figure 3.6, we see that the SIEUFERD system is the only one of the three systems to be able to express all queries in the benchmark. On the other hand, queries that can be expressed in at least one of the other two systems appear to be require more steps in SIEUFERD. A larger part of the difference, however, we attribute to three

[3] http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtFacetBrowserInstallConfig

[4] Official result repository at https://github.com/rhizomik/BESDUI

Table 3.7: Exact interaction steps required to specify each of the 12 queries in the BESDUI benchmark using SIEUFERD. The metrics K, P, and H refer to the number of mouse or keyboard keypresses, mouse aiming operations, and switches between the mouse and the keyboard required for each step, respectively. The speed is the estimated number of seconds required to complete the task based on the indicated standard durations per metric. (Table continued on the next page.)

		KLM Metrics			
		0.2s	1.1s	0.4s	
	Interaction Steps	**K**	**P**	**H**	**Speed (s)**
Task 1: Look for products of type "sheeny" with product features "stroboscopes" and "gadgeteers", and a "productPropertyNumeric1" greater than "450"					
1.	Create and open a perspective from the "product" table	4	3	0	
2.	Make the "propertyNum1", "productfeatureproduct.productfeature.label", and "producttypeproduct.producttype.label" fields visible	9	7	0	
3.	Filter to show only products for which "producttypeproduct.producttype.label" is "sheeny"	4	3	0	
4.	Open the filter toolbox on the "productfeature.label" field	2	2	0	
5.	Click "stroboscopes" and "gadgeteers" and close the filter box	3	2	0	
6.	Insert a calculated field next to "product.label"	2	2	0	
7.	Key the cursor to the calculated column and type the formula "=[propertyNum1]>450 and count([productfeature\label])=2" (field references inserted by arrow keypresses)	17	0	1	
8.	Filter to show only products for which the formula is "true" (use keyboard shortcut since hand is already on keyboard)	6	0	0	
	Total Task 1	**47**	**19**	**1**	**30.7**
Task 2: List products of type "sheeny" with product features "stroboscopes" OR "gadgeteers", and a "productPropertyNumeric1" greater than "450"					
1.	Create and open a perspective from the "product" table	4	3	0	
2.	Make the "propertyNum1", "productfeatureproduct.productfeature.label", and "producttypeproduct.producttype.label" fields visible	9	7	0	
3.	Filter to show only products for which "producttypeproduct.producttype.label" is "sheeny"	4	3	0	
4.	Open the filter toolbox on the "productfeature.label" field	2	2	0	
5.	Click "stroboscopes" and "gadgeteers" and close the filter box	3	2	0	
6.	Insert a calculated field next to "product.label"	2	2	0	
7.	Key the cursor to the calculated column and type the formula "=[propertyNum1]>450" (field reference inserted by single arrow keypress)	8	0	1	
8.	Filter to show only products for which the formula is "true" (use keyboard shortcut since hand is already on keyboard)	6	0	0	
	Total Task 2	**38**	**19**	**1**	**28.9**
Task 3: Get details about product "boozed"					
1.	Create and open a perspective from the "product" table	4	3	0	
2.	Make all the fields visible (use keyboard shortcuts)	8	0	1	
3.	Open the filter box (keyboard shortcut)	1	0	0	
4.	Type "boozed", down, down, space, enter	10	0	0	
	Total Task 3	**23**	**3**	**1**	**8.3**
Task 4: Look for products of type "sheeny" with product features "stroboscopes" but NOT "gadgeteers", and "productPropertyNumeric1" value greater than "300" and "productPropertyNumeric3" smaller than "400"					
1.	Create and open a perspective from the "product" table	4	3	0	
2.	Make the "nr", "propertyNum1", "propertyNum3", "productfeatureproduct.productfeature.label", and "producttypeproduct.producttype.label" fields visible	11	9	0	
3.	Filter to show only products for which "producttypeproduct.producttype.label" is "sheeny"	4	3	0	
4.	Do a custom join operation to join in another instance of the productfeatureproduct table	5	5	0	
5.	Make the "productfeatureproduct2.productfeature.label" field visible	4	3	0	
6.	Filter for "stroboscopes" in "productfeatureproduct"	4	3	0	
7.	Filter for "NOT gadgeteers" in "productfeatureproduct2"	5	4	0	
8.	Insert a calculated field next to "product.label"	2	2	0	
9.	Key the cursor to the calculated column and type the formula "=[propertyNum1]>300 and [propertyNum3]<400" (field references inserted by arrow key presses)	17	0	1	
10.	Filter to show only products for which the formula is "true" (use keyboard shortcut since hand is already on keyboard)	6	0	0	
	Total Task 4	**62**	**32**	**1**	**48**
Task 5: Look for products of type "sheeny" with product features "stroboscopes" and "gadgeteers" and a "productPropertyNumeric1" value greater than "300" plus those of the same product type with product features "stroboscopes" and "rotifers" and a "productPropertyNumeric2" greater than "400"					
1.	Create and open a perspective from the "product" table	4	3	0	
2.	Make the "nr", "propertyNum1", "propertyNum3", "productfeatureproduct.productfeature.label", and "producttypeproduct.producttype.label" fields visible	11	9	0	
3.	Filter to show only products for which "producttypeproduct.producttype.label" is "sheeny"	4	3	0	
4.	Do a custom join operation to join in another instance of the productfeatureproduct table	5	5	0	
5.	Make the "productfeatureproduct2.productfeature.label" field visible	4	3	0	
6.	Filter for "stroboscopes" in "productfeatureproduct"	4	3	0	
7.	Insert a calculated field next to "product.label"	2	2	0	
8.	Key the cursor to the calculated column and type the formula "=[pfp2.pf.label]='gadgeteers' and [propertyNum1]>300 or [pfp2.pf.label]='rotifiers' and [propertyNum2]>400" (field references inserted by mouse clicks)	52	4	4	
9.	Filter to show only products for which the formula is "true" (use keyboard shortcut since hand is already on keyboard)	6	0	0	
	Total Task 5	**92**	**32**	**4**	**55.2**

	Interaction Steps	KLM Metrics 0.2s K	1.1s P	0.4s H	Speed (s)
Task 6: Look for products similar to "boozed", with at least one shared feature, and a "productPropertyNumeric1" value between "427 and 627" (100 more or less than its value for boozed, 527) and a "productPropertyNumeric2" value between "545 and 945" (200 more or less than its value for boozed, 745)					
1.	Create and open a perspective from the "product" table	4	3	0	
2.	Make the "propertyNum1", "propertyNum3", and "productfeatureproduct.productFeature" fields visible	7	6	0	
3.	Do a custom join operation to join the existing instance of productfeatureproduct with another instance of productfeatureproduct on the productFeature field	5	5	0	
4.	Make "product2" then "propertyNum1", "propertyNum3", and "label" fields visible under productfeatureproduct2 (the second instance of productfeatureproduct2)	5	4	0	
5.	Filter "product2.label" to only include the product "boozed" (open filter with keyboard shortcut, type "boozed", click result and close filter popup)	9	8	2	
6.	Insert a calculated field next to "product.label"	2	2	0	
7.	Key the cursor to the calculated column and type the formula "=[propertyNum1]<[product2\propertyNum1]+100 and [propertyNum1]>[product2\propertyNum1]-100" (field references inserted by mouse clicks)	22	4	4	
8.	Copy the formula to the clipboard from the formula bar	2	2	2	
9.	Insert another calculated column next the previous one	2	2	0	
10.	Key the cursor to the new calculated column and paste the formula	3	0	1	
11.	Change "propertyNum1" to "propertyNum3" in each place in the formula in the formula bar, and "100" to "200".	20	6	7	
12.	Filter the two formulas to only include values of "true"	12	0	0	
	Total Task 6	**93**	**42**	**16**	**71.2**
Task 7: Search products whose name contains "ales"					
1.	Create and open a perspective from the "product" table	4	3	0	
2.	Open the filter and type "ales"	5	0	1	
3.	Select all the results (multiple selection via page down + space), then press enter to close the filter popup	6	0	0	
	Total Task 7	**15**	**3**	**1**	**6.7**
Task 8: For the product "waterskiing sharpness horseshoes" list details for all its "offers" by Chinese vendors and still valid by "2008-05-28" plus details for all "reviews" for this product having either "rating1 or rating2" [assumed to mean simply selecting the two fields "rating1" and "rating2"]					
1.	Create and open a perspective from the "product" table	4	3	0	
2.	Open the filter popup and type "wat sha" (assume this makes the right product visible)	8	0	1	
3.	Select the "waterskiing sharpness horseshoes" product and close the filter popup	2	1	1	
4.	Make all the relevant fields visible, including from the "offer" and "review" table instances	16	15	0	
5.	Filter "product.offer.vendor.country" to China only.	4	3	0	
6.	Insert a calculated column next to "offer.validto"	2	2	0	
7.	Key the cursor to the new calculated column and enter the formula "=[validTo]>{2008-05-28}"	16	0	1	
8.	Filter to show only products for which the formula is "true" (use keyboard shortcut since hand is already on keyboard)	6	0	0	
9.	Deactivate "Hide Parent if Empty" on the "offers" table instance (to avoid hiding the product if there are no offers satisfying the constraint--needed if we want to be technically equivalent to the left join in the SQL query example)	2	0	1	
	Total Task 8	**60**	**24**	**4**	**40**
Task 9: For the product "waterskiing sharpness horseshoes" list the "20" more recent "reviews" in "English"					
1.	Create and open a perspective from the "product" table	4	3	0	
2.	Open the filter popup and type "wat sha" (assume this makes the right product visible)	8	0	1	
3.	Select the "waterskiing sharpness horseshoes" product and close the filter popup	2	1	1	
4.	Make the "review" table instance visible and its relevant fields.	7	6	0	
5.	Filter "review.language" on "en"	4	3	0	
6.	Sort descending on "reviewDate" (there is no LIMIT to be specified; the user can simply look at the 20 topmost rows. Explicit limits could be supported by allowing the row_number() window aggregate function in formulas)	2	2	0	
	Total Task 9	**27**	**15**	**2**	**22.7**
Task 10: Get all available information about the author of "Review5481"					
1.	Create and open a perspective from the "review" table	4	3	0	
2.	Show the "review.nr" and "review.person" fields, and all fields under "review.person"	10	9	0	
3.	Filter "nr" to show only Review5481 (use keyboard shortcuts)	10	0	1	
	Total Task 10	**24**	**12**	**1**	**18.4**
Task 11: Look for the "cheapest" and still "valid" by "2008-06-15" "offer" for the product "waterskiing sharpness horseshoes" by a "US vendor" that is able to "deliver" within "3 days"					
1.	Create and open a perspective from the "product" table	4	3	0	
2.	Open the filter popup and type "wat sha" (assume this makes the right product visible)	8	0	1	
3.	Select the "waterskiing sharpness horseshoes" product and close the filter popup	2	1	1	
4.	Make all the relevant fields visible	11	8	0	
5.	Filter "product.offer.vendor.country" to US only.	4	3	0	
6.	Insert a calculated column next to "offer.validto"	2	2	0	
7.	Key the cursor to the new calculated column and enter the formula "=[validTo]>{2008-06-15}"	16	0	1	
8.	Filter to show only products for which the formula is "true" (use keyboard shortcut since hand is already on keyboard)	6	0	0	
9.	Sort ascending on "product.offer.price"	2	2	0	
10.	Filter on "product.offer.deliveryDays" to include "1", "2", and "3"	6	5	0	
	Total Task 11	**61**	**24**	**3**	**39.8**
Task 12: Save in the local computer the information about the vendor for "Offer3499" and, if possible, restrict it to just label, homepage and country and map them to "schema.org" terms name, url and nationality					
1.	Create and open a perspective from the "vendor" table	4	3	0	
2.	Show only the fields "label", "homepage", and "country", as well as "offer" and "offer.nr"	6	5	0	
3.	Filter "offer.nr" to show only Offer3499 (use keyboard shortcuts)	9	0	1	
4.	Hide "offer"	2	2	1	
5.	Key to the "label" column's heading and rename it to "name" (key right to the next column after editing).	7	0	0	
6.	Rename "homepage" to "url" (key right to the next column after editing)	4	0	0	
7.	Rename "country" to "nationality".	12	0	0	
8.	Export to CSV. (Assume this feature exists.)	4	4	0	
	Total Task 12	**48**	**14**	**2**	**25.8**

factors:

- The need, in SIEUFERD, to often click many times to select fields to display in the query. This is because SIEUFERD only shows a small selection of heading-type fields whenever a new relation is introduced into the current query. The latter design decision was made after observations in the formative user study; SIEUFERD displays query results in a nested table layouts by default, and there is only limited horizontal space available for columns. Rhizomer, on the other hand, displays entities in a form layout, showing *all* fields one level from the entity by default. Even though this means that only a few entities can be shown at a time on the screen, there is less of a need to hide and show individual primitive fields in preparation for doing filters and other operations. In Chapter 5 we show how form layouts can be supported from within SIEUFERD. In the future, we might change the policy for default field visibilities based on whether the user is building a query from within a form layout or a nested table layout.

- A lack of range filters in SIEUFERD. Rhizomer features range filters, which makes it easy to specify conditions such as [VALIDTO] > {2008-05-28}. In SIEUFERD, such conditions currently have to be specified as filters on boolean formulas. Range filters could be a very useful addition to SIEUFERD.

- The need, in SIEUFERD, to open a context menu for every operation. Because the KLM metric used in the benchmark does not take the mouse pointer's required travel distance and target size (Fitt's law) into account, it is a bit biased against SIEUFERD's interface, preferring a single click in a sidebar far off the screen over two clicks close to the location of the data being manipulated.

- Some differences in assumptions about query tasks. For instance, in tasks 6, we assume that PRODUCTPROPERTYNUMERIC1 is defined as "100 more or less than its value for boozed", while the Rhizomer evaluation hard-codes the values 427 and 627 into the query.

3.6 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.

Chapter 4

Semantics and Expressiveness

4.1 Overview

In the previous chapter, we presented SIEUFERD's visual query language exclusively from the user's point of view, showing the various possible states of the SIEUFERD *query model* as they would be displayed on the screen during an interactive query building session. In this chapter, we discuss the specific data structure that defines the query model, and how the various operations in our visual query language are mapped to changes in the query model. We then show how a query in the query model is translated to SQL queries for evaluation. Finally, we show how arbitrary SQL-92 queries, via an extended relational algebra, can always be translated to a corresponding query in the SIEUFERD query model.

4.2 The Nested Relational Data Model

In plain SQL, a database query always returns a flat table of results, that is, a bag of tuples of primitive values. As we have seen, however, a query in the SIEUFERD query model can return *nested* results, allowing a greater deal of structure to be communicated in the result of a query. Specifically, we allow queries to produce results from the *nested relational data model* [53, 68], which we define next.

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 field 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 4-1. For our purposes, we will assume that the schema of a relation can define more fields than are actually present in a conforming value; this allows the result of a query to retain information about fields that are currently hidden from view.

Besides using the nested relational data model as a data model for query results, we also its concept of a *schema* in the definition of queries themselves, as will be seen next.

Table 4.1: Properties in the SIEUFERD query model, associated with each field in the nested relational schema that defines a visual query. Along with the core set of properties that are needed to define a database query, we also store various properties that define how the result of the query is presented on the screen during interactive query construction; some examples are shown here. 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 3-1.

	Example User-Defined Presentation Properties
P R	**Label.** Presentation label for the field. Used in nested table headers, in the field selector, and as part of the syntax for referencing fields from formulas. Defaults to the original technical name of the column or table in the underlying database.
P R	**LabelTextStyle.** Text style (font, size, etc.) for the field's presentation label.
P	**ValueTextStyle.** Text style for data values associated with this field.
	Example Auto-Measured Presentation Properties
P	**ValueWidth.** The column width to use for this field when rendering results in a table layout. Typically an average or maximum of the visual widths of individual data values, up to some limit, depending on the data type.
P	**DecimalPlaces.** For numeric fields, the number of decimal places to show in formatted data values. The default heuristic tries to ensure that every numeric data value will be shown with up to four of its significant digits.
P	**SampleSize.** The number of unique data values that were used the last time measured properties such as ValueWidth and DecimalPlaces were calculated. Used to decide when recalculations should happen.
	Query Definition Properties
P R	**Visible.** Boolean indicating whether this field should be visible in the result layout.
P R	**Filter.** An optional 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, and handles the most common kind of join without requiring the use of formulas and filters.
P	**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.** Boolean indicating projection policy for primary key fields. 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.** Boolean indicating if an inner join rather than a left join should be used between this relation and its parent. Set automatically by the filter UI, but can be overridden.
R	⊰ **OneToMany.** Read-only boolean, set automatically to indicate to the user the presence of a one-to-many relationship at this relation. Always on except for relations whose JoinedOn conditions provably ensure that there would be only one nested child tuple per parent tuple, e.g. for joins on the child relation's primary key.

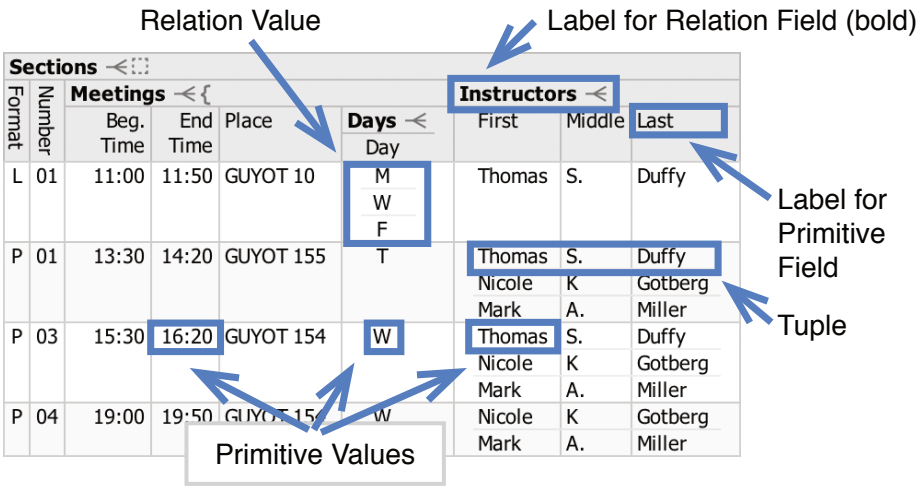


Figure 4-1: Terminology of the nested relational data model, illustrated on a nested table layout.

4.3 The SIEUFERD Query Model

A visual query is modeled as a nested relational schema that has been annotated with query- and presentation-related properties on each field. The query-related properties, and some examples of presentation-related properties, are shown in Table 4.1. The annotated schema, which we will refer to as the SIEUFERD *query model*, fully defines both the SIEUFERD query to be executed and how its results should be rendered on the screen.

We have defined the SIEUFERD query model so as to maintain a very particular relationship between the structure of a query and the structure of its result. This is what allows us to provide a single direct manipulation interface through which the user can *edit the query* by manipulating the *result* of the query. When a SIEUFERD query is executed, returning a nested relational result, *the schema of the nested query result is identical to the query model schema that defined the query in the first place*. The correspondence between the structure of the query and the structure of its result 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.

The concept of encoding a database query in the schema of its own result is a key idea in our design, and typically not how other query languages work. For instance, the abstract syntax tree of a SQL query has no well-defined relationship with the schema of the query's table of results. In XQuery, the query and the result are both defined in the same data model (XML), but the structure of the query is still not guaranteed to be reflected in the schema of the result. The encoding of complex multi-block queries into simple annotations on the schema of their own results is possible in part because we chose a nested data model for results, as opposed to the flat tabular model of SQL results.

4.3.1 Encoding Examples

We return to a few of the visual query examples from Section 3.2.2, explaining how each would be encoded in the query model's data structure. The *property* concept refers to the schema field properties that were defined in Table 4.1.

Table instantiation. Our first example was a simple selection of tuples from a table in the database:

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. The term *technical name* here refers to the string that is used to identify a field or column in generated SQL queries. The separate property LABEL holds the name that is actually displayed in the result layout for presentation purposes. While LABEL defaults to the technical name of the field or column, the user can edit the cell containing the label to change the label to something more human-readable if desired.

In the example above, all fields have the VISIBLE property set to TRUE, while the FILTER, SORT, JOINEDON, COLLAPSEDUPLICATEROWS, and HIDEPARENTIFEMPTY properties are cleared. The read-only ONETOMANY property (≺) is not set by the user, but is set automatically on the COURSES relation as an indication that more than one tuple may be returned for that relation.

Note that the INSTANTIATEDTABLE, COLLAPSEDUPLICATEROWS, HIDEPARENTIFEMPTY, and ONETOMANY properties only apply to relation fields, while the JOINEDON and COLUMNDEFINITION only apply to primitive fields, as indicated in Table 4.1.

Nesting and equijoins. In the next example, a nested relation field READINGS is added to the COURSES relation after the latter relation's existing primitive fields. The READINGS relation has the INSTANTIATEDTABLE property set to reference the READINGS table, and includes its own primitive child fields each with the COLUMNDEFINITION property set to a corresponding column in the READINGS database table. The JOINEDON property of the READINGS\COURSE_ID field is set to reference the COURSES\ID field, denoting an equijoin condition. The presence of an equijoin condition is indicated by the join icon (⋈). To avoid clutter, the referenced field on the other side of the join condition is not displayed in the table header, but can be seen in the field selector.

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

Hidden fields. In the previous two examples, all fields had the VISIBLE property set to TRUE. When a field is hidden, it remains present in the in the query model's annotated schema, but has the VISIBLE property set to FALSE. The primary visual effect is is that the fields in question are now hidden from view:

courses ≺				readings ≺	
title	may_pdf	may_audit	exam_type	author_name	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 VISIBLE property usually also determines whether data is retrieved for a particular field or not, with a few exceptions (defined later) involving the SORT and COLLAPSEDUPLICATEROWS properties.

Note that it is permissible for a non-VISIBLE relation to have VISIBLE descendants. This allows a relation to be hidden and unhidden without altering the visibility state of its child fields. The visibility state of child field of a non-VISIBLE relation may also be semantically significant in the case where the latter relation is referenced from an aggregate function and has HIDEDUPLICATEROWS enabled, as hiding and showing a field, even when the child of a hidden relation, may change the number of tuples in the hidden relation for the purposes of

evaluating the aggregate. Thus, the VISIBLE property does not mean "visible on the screen", but rather "this field would be visible on the screen if its parent was visible on the screen".

Sorting. The presence of a sort icon, either ascending ($\equiv\downarrow$) or descending ($\equiv\uparrow$), and possibly with a subscript ($\equiv\uparrow_{123}$) indicating the position of the sort term in the underlying `ORDER BY` clause, fully defines the contents of a field's SORT property. A sort icon without a subscript has position 1. In the following example, the AREA and CODE fields both have their SORT property set to an ascending sort of position 1, while the COURSES\TITLE field has the SORT property set to an ascending sort of position 2:

courses							
id	area_id	area ⋈ id	area title	area code	title $\downarrow_2$	may_pdf	may_audit
177	2	2	Literature and the Arts	LA	Comedy	Y	Y
56	2	2	Literature and the Arts	LA	Roman Art	N	Y
845	2	2	Literature and the Arts	LA	Russian Drama	N	N
1795	4	4	Social Analysis	SA	American Politics	Y	Y
3921	4	4	Social Analysis	SA	Judicial Politics	Y	Y
2566					Junior Seminars	N	N

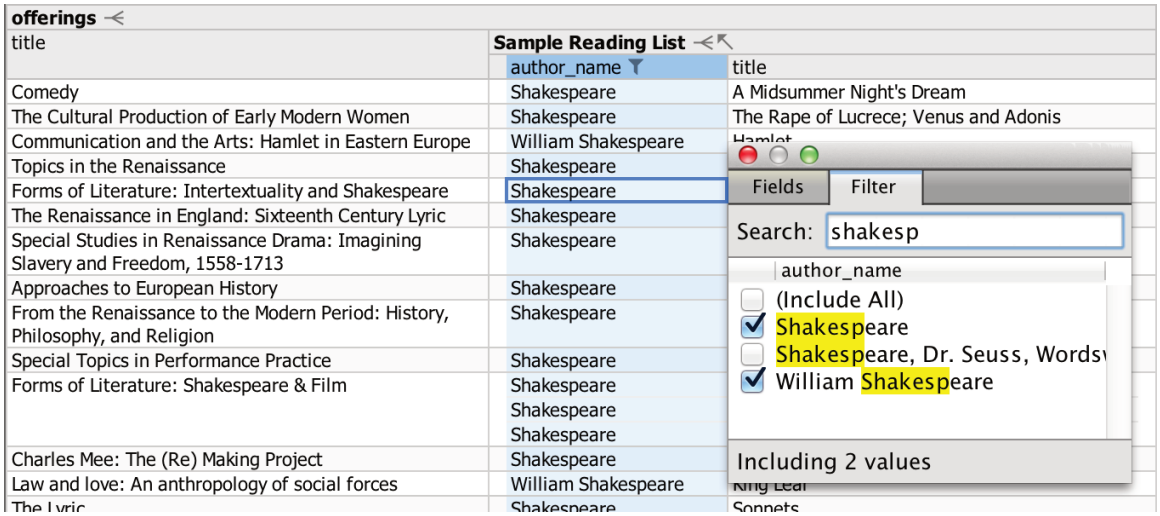
Formulas. For calculated fields, a field's COLUMNDEFINITION specifies a formula over other fields in the query model instead of a column in its parent relation's instantiated database table. In the following example, the DURATION field has COLUMNDEFINITION set to the formula =SUM([DURATION]), indicated with the formula icon (f_x). Only primitive fields have a COLUMNDEFINITION property.

=sum([duration])

courses							
title	f_x total duration	sections type	sections num	meetings day	meetings start	meetings end	meetings 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

Filter. If a field has a filter defined on it, the state of that filter is stored in the FILTER property, with its presence indicated with the filter icon ($\blacktriangledown$). Seeing the complete state of the filter requires opening the filter popup. Internally, the FILTER property simply holds another formula of the same form that is used for calculated fields, with some restrictions.

The formula must (1) return a boolean value, (2) not contain aggregate functions, (3) reference only the filtered-on field or, for filters on relations, its primitive child fields, and (4) be of a form that can be edited in filter popup UI. None of these restrictions affect the expressiveness of the visual query language, as the user can always create a new calculated field based on an arbitrarily complex formula, boolean or otherwise, and then filter on that calculated field instead. In the following example, the AUTHOR_NAME column has the FILTER property set to the boolean formula =[AUTHOR_NAME] IS "SHAKESPEARE" OR [AUTHOR_NAME] IS "WILLIAM SHAKESPEARE".



Other boolean properties. The state of the remaining properties, COLLAPSEDUPLICATEROWS and HIDEPARENTIFEMPTY, is indicated directly with corresponding icons ({,↖). In the following example, the COLLAPSEDUPLICATEROWS and HIDEPARENTIFEMPTY properties are both set to TRUE on the SECTIONS relation field.

offerings

title	sections	
	format	status
Introduction to African-American Literature: Harlem Renaissance to Present	S	X
Topics in African American Religion	P	X
Topics in African American Literature	P	X
The Caribbean in the American Imagination	S	X
Introduction to Anthropology	L	X
	P	X

4.4 Operations on the Query Model

We now discuss how the various high-level UI operations supported by SIEUFERD's visual query language map to modifications on the underlying query model. Every query-related action acts purely as a modification to the current instance of the SIEUFERD query model. No modifications are made to the underlying relational database, and no additional

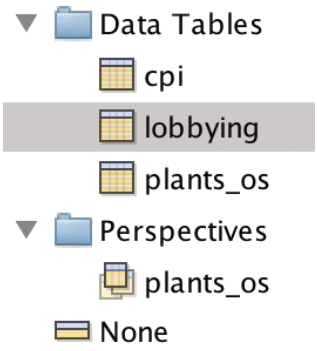


Figure 4-2: The query list, which includes one automatically generated base query for each table in the database (DATA TABLES), an empty query not instantiating any table (NONE), and queries previously created by the user (PERSPECTIVES). The query list is used in the initial selection of a template for a new query, as well as in the JOIN dialog.

state need be kept between query actions beyond basic schema metadata such as the list of available tables, table columns, and known foreign key relationships. Using operations described below, the user can reach all valid states of the SIEUFERD query model.

Query list. Before entering the main query building interface, the user selects an existing query to use as a template for the new one. The available options include one automatically generated base query for each table in the database, an empty query not instantiating any table, as well as any queries previously created by the user. See Figure 4-2. The automatically generated base queries consist of a single relation instantiating the table in question, with primitive child fields selecting each of the table's columns, as in our very first example query. The empty template query consists of a single relation with the INSTANTIATEDTABLE property cleared, with no child fields. The latter template is useful primarily for dashboard-style queries, where multiple independent subqueries are shown side by side as nested relations under a singleton root relation.

Once the user has created a new query, all subsequent query actions, with the exception of formula editing and label editing, are initiated from the *context menu*. We have seen the context menu in the previous chapter; we show it again in Figure 4-3 for reference as we go through each of its actions. The editing of formulas and labels can be done directly in cells without opening the context menu, modifying the COLUMNDEFINITION and LABEL properties, respectively, in the underlying query model.

The context menu can be opened on any field or multiple selection of fields in the result layout. When describing context menu actions below, we assume that only a single field has been selected, although many of the actions can be defined to have meaningful behavior for a selection of multiple fields as well.

Field selection. The selection of fields in the field selector, opened with the FIELDS action in the context menu, lets the user directly modify the boolean VISIBLE property of each field in the query model. See Figure 4-4. The HIDE action in the context menu is a shortcut that sets the VISIBLE property to FALSE for the selected field. The field that the context menu was opened on determines which relation is shown as a root in the field selector, as well as which field is initially selected in the field selector. This lets the user

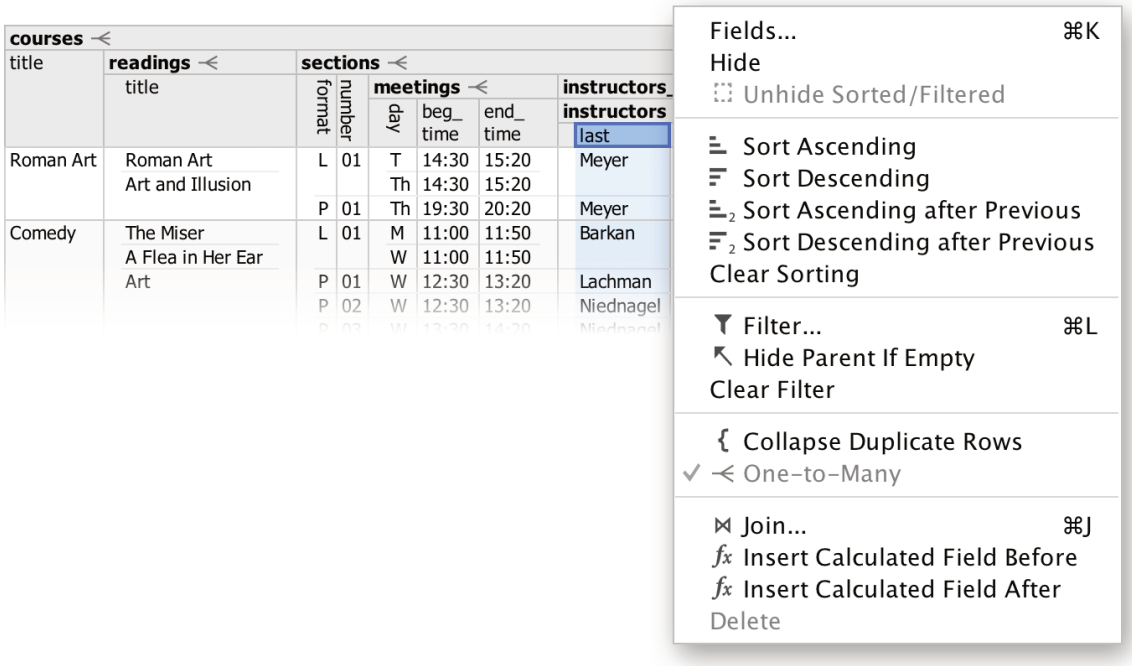


Figure 4-3: The context menu, which serves as a starting point for all query-related actions. The context menu can be opened on any field or multiply selected set of fields.

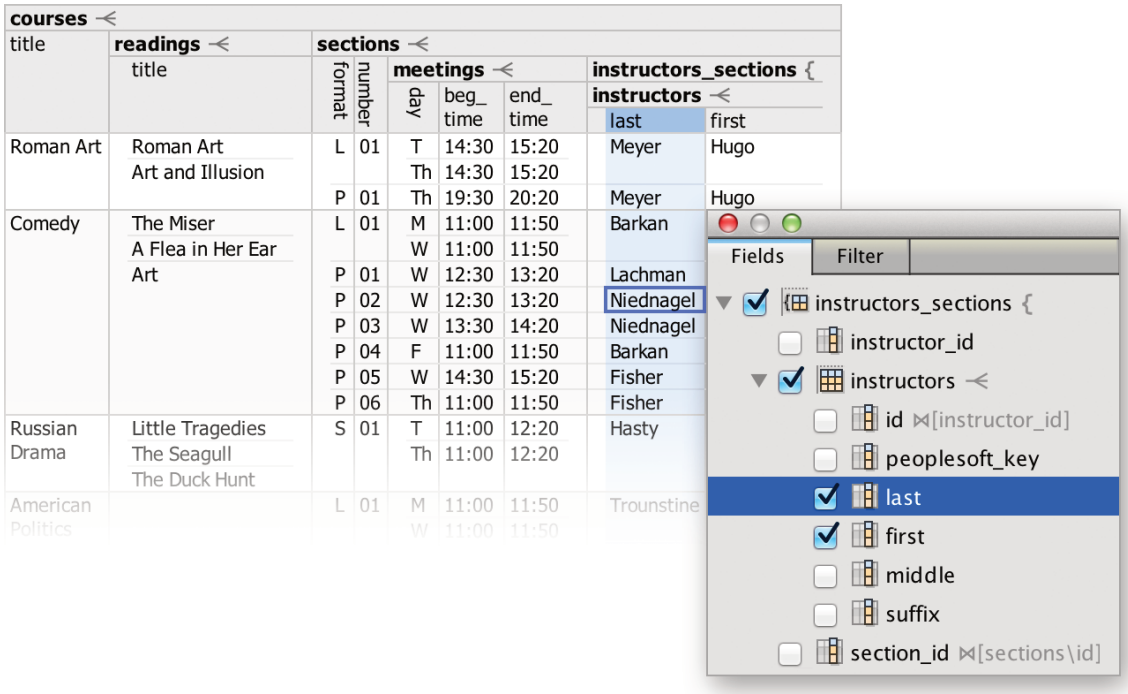


Figure 4-4: The field selector. The user invoked the FIELDS action from the context menu while the LAST field was selected, so the field selector shows the latter field as initially selected along with its visible and non-visible sibling fields.

quickly open the relevant part of the schema tree from the result layout instead of having to expand the entire tree in the field selector from the query's root relation.

The exact logic for determining which relation to show as the root in the field selector, based on the field or fields that were selected in the result layout, can be tweaked for usability. The tree of fields is automatically expanded so that the selected field or fields can be shown as selected when the field selector is initially opened.

One additional aspect of the field selector is the automatic suggestion of nested joins over foreign key relationships, based on the InstantiatedTable property of each relation. Such joins are represented in the query model in the same way as manually added joins (see below), but initially have their nested relation's Visible property set to false. The latter ensures that the added relation has no semantic impact on the user's query unless the user decides to make it Visible.

The Unhide Sorted/Filtered/Referenced action is a shortcut for unhiding currently invisible fields that are being sorted on, filtered on, or referenced from a formula, respectively. This action is normally only visible in the context menu when actually applicable, though we have shown its position in Figure 4-3 for completeness.

Sorting. The context menus' various sorting-related actions lets the user reach all meaningful states of the query model's Sort property. Sort Ascending/Descending sets the Sort property on the selected field, with ordinal 1, and clears it on all sibling fields. Sort Ascending/Descending after Previous sets the Sort property on the selected field without clearing the Sort property on sibling fields, instead using an ordinal one higher than that of the sibling with the previously highest sort ordinal. Clear Sort, if invoked on a relation field, clears the Sort property on all of that relation's child fields. If invoked on a primitive field, it instead clears the Sort property for that field and any siblings with a higher sort ordinal.

Filter. The Filter action opens the filter popup to let the user define a boolean condition for the query model's Filter property for the field in question. Setting or resetting a filter also automatically sets the HideParentIfEmpty property to true or false, respectively, for all ancestor relations of the filtered field, although the this property can also be modified directly using the Hide Parent If Empty action in the context menu. Clear Filter is a shortcut that lets the user clear a filter without opening the filter popup.

Collapse Duplicate Rows. This context menu action allows the user to directly modify the boolean CollapseDuplicateRows property on the selected relation field. If a primitive field is selected, the action applies to the parent relation.

One-to-Many. This context menu entry is always disabled, serving only as an indication of the state of the read-only OneToMany property and as a legend for the one-to-many icon ($\prec$).

Join. The Join action, described in the most general way, allows the user to select a relation in the current query and add to it, as a new nested child relation, a copy of an entire other SIEUFERD query, optionally specifying equijoin conditions between the parent relation and the new child relation. The selected equijoin conditions are then encoded in the JoinedOn property of the new child relation's primitive child fields. The other query to copy into the current query, and the equijoin conditions, are selected in a dialog box, shown in Figure 4-5. The list of other queries available to copy is the same as that shown when initially creating a new query (Figure 4-2).

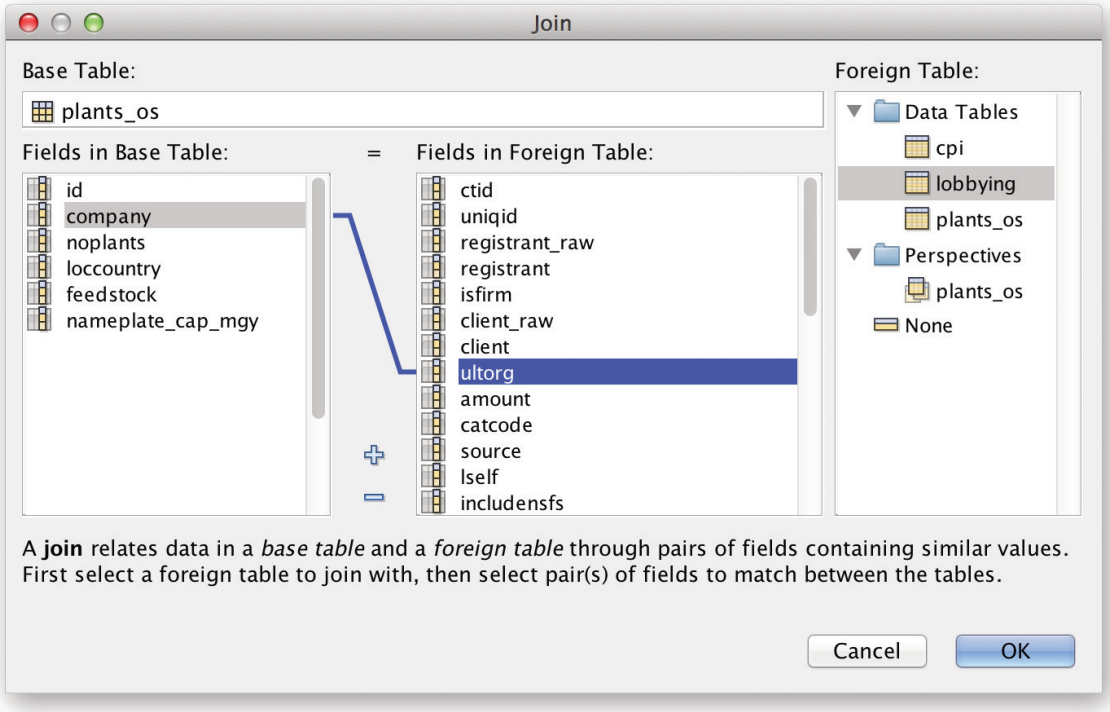


Figure 4-5: The JOIN dialog box, which is used to define custom equijoin conditions against an arbitrary new table instance. The query list on the right is the same as that which was shown in Figure 4-2.

Most commonly, the user will use the JOIN action to instantiate a single new table and join it against a parent relation field in the current query. In this case, the other query to copy into the current query is one of the template queries that our system creates automatically for each table in the database.

For joins over foreign key relationships that are declared at the database level, use of the JOIN action is not necessary; such joins will automatically be added and available in the field selector by default.

Insert Calculated Field. The INSERT CALCULATED FIELD BEFORE/AFTER actions add a new primitive field as a sibling of the selected field, with the property COLUMNDEFINITION set to a default empty formula. The formula can then be edited directly in the query result layout, or using the formula bar.

(For completeness, we also have an action that allows the user to add a calculated field as a child of a selected relation. This action is only shown, and needed, in the rare case that the selected relation has no existing children at all.)

Delete. The DELETE action can be used to delete primitive fields that have COLUMNDEFINITION set to a formula, or any relation field. It is disabled for primitive fields that have COLUMNDEFINITION set to refer to a database table column. The latter kind of column may be hidden, but not deleted.

4.5 Query Evaluation

We now explain the steps required to translate a query in the SIEUFERD query model to a set of SQL queries that can be executed on a relational database backend in order to evaluate the SIEUFERD query. We first explain how a query in a simplified version of the SIEUFERD query model can be translated to and evaluated as a single SQL query. We then explain how to extend the simplified query model to support the retrieval of nested relational results. Finally, we explain how a query in the fully general version of the SIEUFERD query model can be rewritten as a query in the simplified query model.

4.5.1 Simplified Query Model

Definition. We define the *simplified* SIEUFERD query model to be equivalent to the general query model, except with the following restrictions on queries:

- The VISIBLE property is set to FALSE for all relations except the root. This means that all queries will return flat tabular results only.
- The SORT property is always cleared.
- For relations with COLLAPSEDUPLICATEROWS disabled, all of the INSTANTIATEDTABLE's primary key fields are VISIBLE.
- Only VISIBLE primitive fields may have JOINEDON defined.
- Only primitive fields may have FILTER defined.

- Each relation contains at most one aggregate function. That is, each relation field may have at most one primitive child field with ColumnDefinition set to a formula containing an aggregate function.
- Formulas (set via ColumnDefinition) contain no inward references except to Visible primitive fields in the formula's parent relation's immediate child relations. An *inward reference* in a formula means a reference to descendant of the formula's field's parent relation that is not a sibling of the formula's field.
- Formulas contain no outward references. An *outward reference* means a reference to a field that is not a descendant of the formula's parent relation.

Every query in the simplified query model is also a valid query in the general query model, with equivalent semantics. The simplified query model is equivalent in expressiveness to the general query model, except that the simplified query model may only retrieve flat tabular results with an unspecified order. We will lift the latter restriction in the next subsection. In all cases we assume that queries contain no user errors such as circular dependencies in formulas or join conditions.

Translation to SQL. In the simplified query model, each relation field and its subtree can be considered an independent query that corresponds to a single SELECT statement in SQL. To produce a SQL query for a given relation field, each of the relation's child relations is first recursively translated into a SELECT statement of its own. The parent relation then builds its own SELECT statement, nesting the SELECT statements of its child relations in its own FROM clause. The recursion stops when the translation reaches a relation with no child relations. Note that the nested SELECT statements are not correlated subqueries, as they do not refer to any columns defined in the outer query.

The general form of the SELECT statement generated for each relation is as follows:

```
SELECT projected_expressions
FROM
  instantiated_table,
  inner_joined_subquery_1, ..., inner_joined_subquery_N
  LEFT JOIN left_joined_subquery_1 ON left_join_condition_1
  ...
  LEFT JOIN left_joined_subquery_M ON left_join_condition_M
WHERE inner_join_condition_1 AND ... AND inner_join_condition_N
  AND scalar_filter_conditions
GROUP BY group_by_column_names
HAVING aggregate_filter_conditions
```

The various clauses are constructed as follows:

- SELECT clause: One column expression is generated for each Visible primitive child field of the relation being translated, based on the formula or column reference in the ColumnDefinition property. Expressions may reference columns in the InstantiatedTable or any of the nested SELECT queries generated for child relations in the FROM clause.

- `FROM` clause: The `FROM` clause first specifies the Cartesian product of the InstantiatedTable, if any, plus a nested `SELECT` statement for each translated child relation for which HideParentIfEmpty is true. The `FROM` clause then includes a nested `SELECT` statement in a `LEFT JOIN` for each translated child relation for which HideParentIfEmpty is false. For each child relation, the JoinedOn property of each of that child relation's primitive child fields is translated into an equijoin constraint that is included either in the `WHERE` clause or, for left joins, the `LEFT JOIN`'s `ON` clause.

  If the equijoin condition of one `LEFT JOIN` clause references fields in a subquery introduced via another `LEFT JOIN` clause, then the left joins are automatically ordered so as to satisfy the dependency. The latter situation may occur if a relation is joined on a calculated field with an inward reference to a sibling relation. It is a user error for a circular dependency to exist in these dependencies, analogous to MySQL's "Cross dependency found in OUTER JOIN" error.

- `WHERE` clause: In addition to the inner join conditions mentioned in connection with the `FROM` clause, the `WHERE` clause includes Filter conditions set on the translated relation's non-aggregate primitive child fields. A primitive field is said to be an *aggregate* field if its ColumnDefinition property is set to a formula containing an aggregate function, or to a formula that references an aggregate sibling field.

- `GROUP BY` clause: This clause lists all the non-aggregate fields from the `SELECT` clause.

- `HAVING` clause: This clause lists Filter conditions set on the translated relation's aggregate primitive child fields.

4.5.2 Nested Relational Results

The previously generated SQL queries only generate flat tabular results. A key feature of our visual query language is the ability to generate nested relational results, whether to visualize aggregate inputs during query construction or as a way to generate complex form or report layouts. The semantics of queries returning nested relational results is most easily explained by assuming the presence of a concatenating aggregate function such as PostgreSQL's json_agg[1]. This allows each query in the SIEUFERD query model to still be translated into a single SQL query, but now returning a nested relational result encoded as a JSON object.

We define the *simplified query model with nested results* to be equivalent to the simplified query model, but now allowing the Visible property to be set to true on relation fields other than the root, allowing the Sort property to be set on primitive fields, and allowing more than one aggregate field in each relation.

To translate a query in the simplified query model with nested results to the plain simplified query model, we follow the following steps:

[1] https://www.postgresql.org/docs/9.6/static/functions-aggregate.html

1. Enclose the root relation of the original SIEUFERD query in a new otherwise empty root relation. This will ensure that the translated SQL query returns a single tuple containing a single JSON object, avoiding a the need for a special case for ordering and retrieving tuples for from the query's original root relation.

2. In depth-first order, for each VISIBLE non-root relation field in the query, add a new VISIBLE primitive sibling field containing an aggregate formula JSON_AGG(JSON_BUILD_OBJECT($a_1, ..., a_n$) ORDER BY $b_1, ..., b_m$), where $a_1, ..., a_n$ are the VISIBLE primitive child fields of the relation being processed, and $b_1, ..., b_m$ are the sort terms defined by those primitive child fields' SORT property. Finally, set the relation's VISIBLE property to FALSE.

3. In depth-first order, eliminate cases where a relation contains more than one aggregate field by replacing each aggregate field with a formula containing an inward reference to a new child relation that calculates only that aggregate by itself. The new child relation is a copy of the original relation, but omits the other aggregate fields, and has the JOINEDON property set on its non-aggregate primitive fields to equijoin against the original relation's `GROUP BY` fields.

4. When executing the generated SQL, interpret each returned JSON array as a nested relation, and return the nested relational result.

The above method of retrieving nested relational query results is conceptually simple, but assumes that the database backend supports the JSON_AGG function. Our actual implementation uses a more portable approach, retrieving the nested results of a single SIEUFERD query using multiple SQL queries, in a manner similar to that of SilkRoute [40]. We have not examined the difference in performance between the two approaches; this would be interesting future work.

4.5.3 Desugaring the General Query Model

We have shown how queries in a simplified version of the SIEUFERD query model can be translated to SQL for retrieval of either flat or nested results. We now outline how a query in the general query model can be translated to a query in the simplified query model, lifting the remaining simplifying restrictions.

The translation from a query in the general version of the SIEUFERD query model to a query in the simplified query model is done through the following rewriting steps:

1. Make any primitive field that has SORT enabled on it VISIBLE.

2. Clear any filter on a relation field, instead inserting a new non-VISIBLE calculated field as a child of the relation, defined by the boolean formula that defined the original relation filter. Set a filter on the calculated field to only include values of TRUE.

3. In depth-first order, for any relation field with SORT enabled on it, add a new primitive sibling field for each VISIBLE primitive child field of that relation, with COLUMNDEFINITION set to a formula consisting of a single reference to that primitive

child field. Replace the SORT on the relation field with a sort on the added primitive sibling fields.

4. Rewrite formulas such that every non-sibling reference is either to a field in the immediate parent relation or an immediate child relation of relation containing the calculated field. References that originally traversed multiple relations can be rewritten to single-level references by adding non-VISIBLE formula fields in the intermediate relations, each formula containing a single reference to the next level.

5. In depth-first order, for any formula referencing a non-VISIBLE primitive field in a child relation, make the referenced field VISIBLE. Do the same for any non-VISIBLE field with JOINEDON defined.

6. Eliminate outward references in formulas by the method outlined on page 56.

Note that the translations done in this section have no impact on the way results are presented to the user on the screen. The translations are done only as a step in the algorithm for generating SQL queries. For presentation purposes, it is always the original, untranslated version of the query that is used to determine which fields are displayed, what labels and icons are shown in the result header, and so on.

4.6 Expressiveness

Like Liu and Jagadish [71], 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 [43, 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 4.1.

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 (i.e. a table in the database), 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.

- *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 $\{($false$), ($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 4-6 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, 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

courses

3rows			course_title	dept1	cnum1	dept2	cnum2	dept3	cnum3
i	dept f_x	cnum f_x							
1	ANT	206	Human Evolution	ANT	206	EEB	306	GEO	208
2	EEB	306							
3	GEO	208							
1	APC	199	Math Alive	APC	199	MAT	199		
2	MAT	199							
3									
1	CLG	108	Homer	CLG	108				
2									
3									
1	WWS	313	Peacemaking	WWS	313	POL	387		
2	POL	387							
3									

```
[dept]  =if([i] = 1, [dept1], if([i] = 2, [dept2], [dept3]))
        =union([dept1], [dept2], [dept3])

[cnum]  =if([i] = 1, [cnum1], if([i] = 2, [cnum2], [cnum3]))
        =union([cnum1], [cnum2], [cnum3])
```

Figure 4-6: 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.

field, or by using the JoinedOn property instead of filters on formula fields.

Chapter 5

Result Layouts

5.1 Introduction

So far, we have discussed the visual query language that allows users to express arbitrary database queries, but not how the results of those queries are actually formatted to be displayed. The latter is the topic of this chapter.

An important class of visualizations in everyday business use consists of the table-, form-, and report-style views found in most tailored CRUD applications (see Chapter 1). The data being displayed is typically *structured*, meaning that each value has an associated type and label in a schema. Furthermore, the data frequently needs to be presented in a *nested* manner, because of the need to visualize one-to-many relationships between entities in the database. For instance, when users request "a list of employees, grouped by department" or "all information about a customer, including associated support tickets and a list of open orders," what is being displayed is a structured nested view of the underlying relational data. This is exactly the kind of data that is produced by SIEUFERD's visual query language, as well as by many previously proposed visual query systems [50, 38, 107, 31, 72, 81, 63, 14, 108, 66, 86, 10, 2, 47, 3, 27, 83, 74, 25].

The problem lies in the hard manual labor involved in formatting nested data for display. Traditionally, a software developer has to define low-level details of the visual layout: the location of labels, the dimensions of text fields, the width of table columns, the organization of form fields into columns on a page, and so on. Besides making the development of new CRUD applications costly, requiring this kind of manual formatting work would be unacceptable in an interactive query system like SIEUFERD, since every query manipulation action can change the schema of the query result and thus require the output to be reformatted.

In this chapter, we present a layout management algorithm that fully automates the display of structured nested data using visual idioms seen in traditional hand-designed database UIs: tables, multi-column forms, and outline-style indented lists. The system gathers simple statistics about fields in the input schema, and uses these to make layout decisions which are then applied uniformly across tuples in each input subrelation.

Our algorithm is illustrated in Figure 5-1. Layouts produced by our algorithm are hybrids between two existing types of layouts: nested table layouts and outline layouts. The

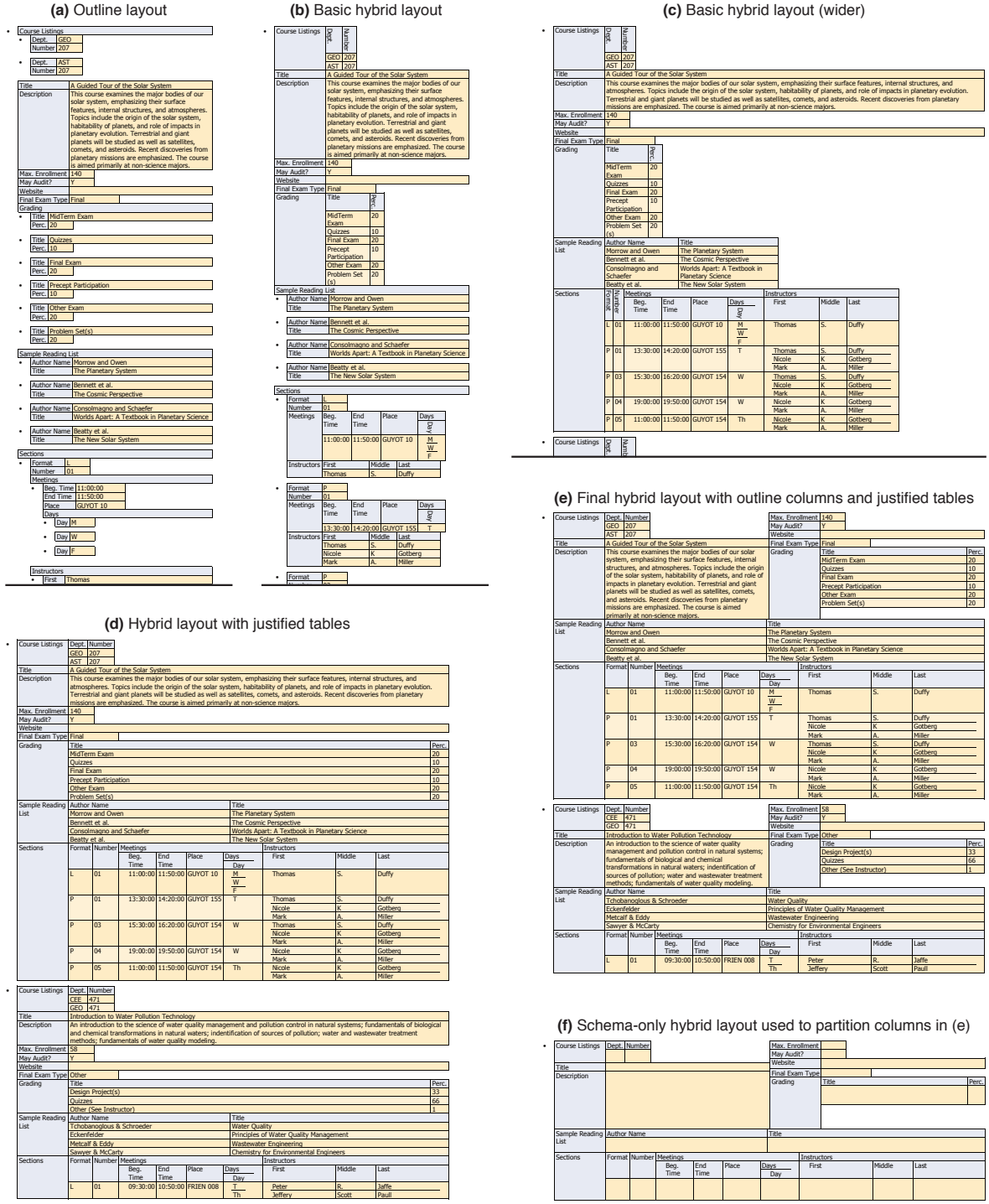


Figure 5-1: We illustrate our algorithm by enabling its features one by one and producing successive layouts of the data from Figure 5-3. All layouts are at the same scale. (a) is a basic outline layout; this layout renders tuples in relation values as indented bullets, stacks the fields of each tuple vertically, and puts labels to the left of primitive values and above relation values. (b) and (c) show basic hybrid layouts, at two different widths, that use the outline layout at the first level but switch to nested table sublayouts wherever a table can fit within the available horizontal space. (d) justifies the columns of the table sublayouts to fill the remaining available horizontal space. (e) adds columns to the outline sublayout to use horizontal space more efficiently. (f) is a schema-only layout generated by the algorithm to calculate ideal break points for the columns in (e).

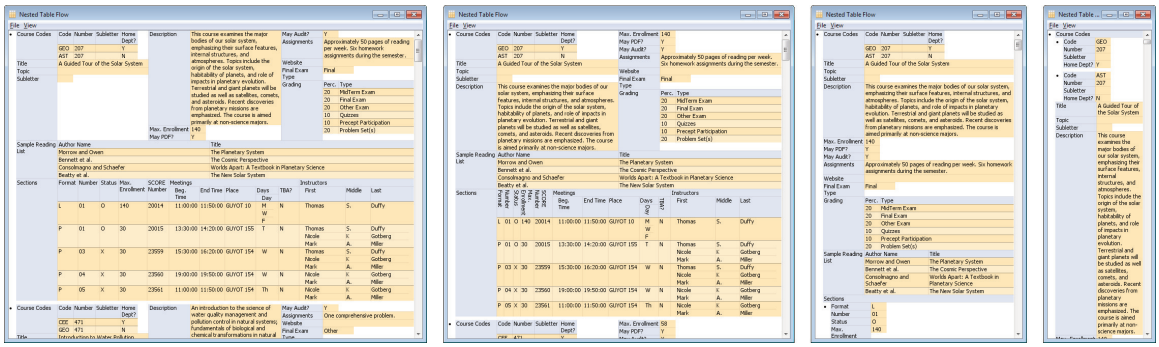


Figure 5-2: Interactive adaptation of the layout of the data to be displayed, based on the available horizontal space in an on-screen window.

nested table layout is the layout that has been seen many times in the previous two chapters; it arranges tuples vertically, and the fields within each tuple horizontally. The outline layout, on the other hand, arranges both tuples and fields vertically, in an indented bullet list. See Figure 5-1(a). The basic hybrid layout, shown in Figures 5-1(b), (c), and (d), replaces the outline layout with a nested table layout for specific relation fields in the schema, wherever such replacement can be done without making the layout too wide for the available screen or page size. This leads to more compact layouts without introducing horizontal scrolling. Finally, our algorithm reclaims additional wasted space by allowing narrow fields in the outline layout to be arranged in columns, as shown in Figure 5-1(e). The ideal placement of column breaks, as well as the decision to use an outline or table layout for a given relation field, is done using an idealized layout produced using average lengths of each field, shown in Figure 5-1(f). In all cases, the order of fields in the layout is kept the same as in the schema of the input data.

We compare our hybrid layout system with pure outline and nested table layouts with respect to space efficiency and readability, the latter with an online user study on 27 subjects. In terms of screen area, our hybrid layouts are 3.9 and 1.6 times more compact on average than outline layouts and horizontally unconstrained table layouts, respectively. This increases the amount of data that can be shown in a given area without scrolling or pagination. Furthermore, our user study shows hybrid layouts to be as readable as table layouts even for large datasets that do require scrolling.

Our Java-based implementation of the layout algorithm, which is now a part of the larger SIEUFERD system, can produce layouts based on data in any well-formed XML document, or based on data retrieved from a relational database using the visual query interface that was described in Chapter 3. For XML document inputs, a suitable schema will be derived automatically. Generated layouts can either be rendered on screen through a custom Swing component, as in Figure 5-2, or printed as vector graphics to paper or fully text-searchable PDF files using the Java Printing API, as was done to produce the other figures in this chapter. In either case, layouts adapt automatically to fit within the available horizontal space. Figure 5-2 shows this demonstrated interactively by resizing a window containing our custom Swing component.

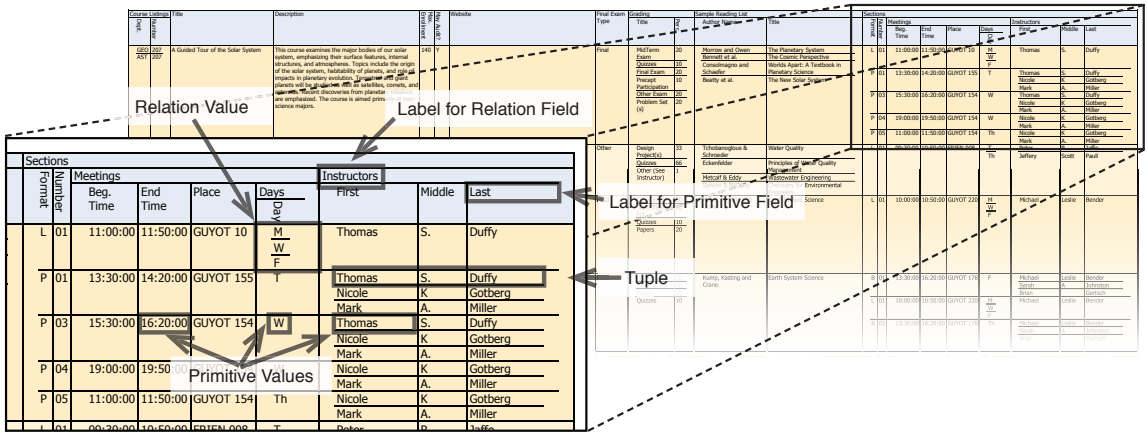


Figure 5-3: The nested table layout is the most common way to visualize a nested relation. Our version of this kind of layout, shown here, is used as a base case in our recursive layout algorithm. Here, we show nested relational data generated from an academic course catalog in nested table style, with one of the nested columns enlarged to show terminology. Nested table layouts arrange tuples in the vertical direction and the fields of each tuple in the horizontal direction, with all field labels collected in a header on top.

5.2 Layout Algorithm

We now describe our layout algorithm. For presentation purposes, we start by discussing basic nested table and outline layouts, then discuss hybrid table/outline layouts, and finally show the complete steps to produce the layouts automatically.

The purpose of the algorithm is to produce a compact but readable view of nested input data that conforms to a non-recursive schema, or, more precisely, nested relational data. Whenever possible, the regularity of the input data's schema should be used to maximize the readability of the data. For instance, a sequence of tuples with similar fields should be rendered as a table, with proper column headings describing the name of fields in the schema. We will not require user input informing layout decisions beyond what can be derived from the data itself.

5.2.1 Nested Relations and Nested Table Layouts

Our algorithm operates on nested data conforming to a non-recursive schema. We have chosen the nested relational model as the concrete data model for our implementation, since it tends to lead to simple tree traversal code. It is equally feasible to use a different nested data model such as that of XML, and in connection with the user study, we did write a routine for importing arbitrary XML documents.

As in the previous chapters, we define the nested relational data model as follows: A *value* is either a *primitive* or a *relation*, where a relation is defined as a set of *tuple*s, each containing a set of *field*s identified by *label*s, each field 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.

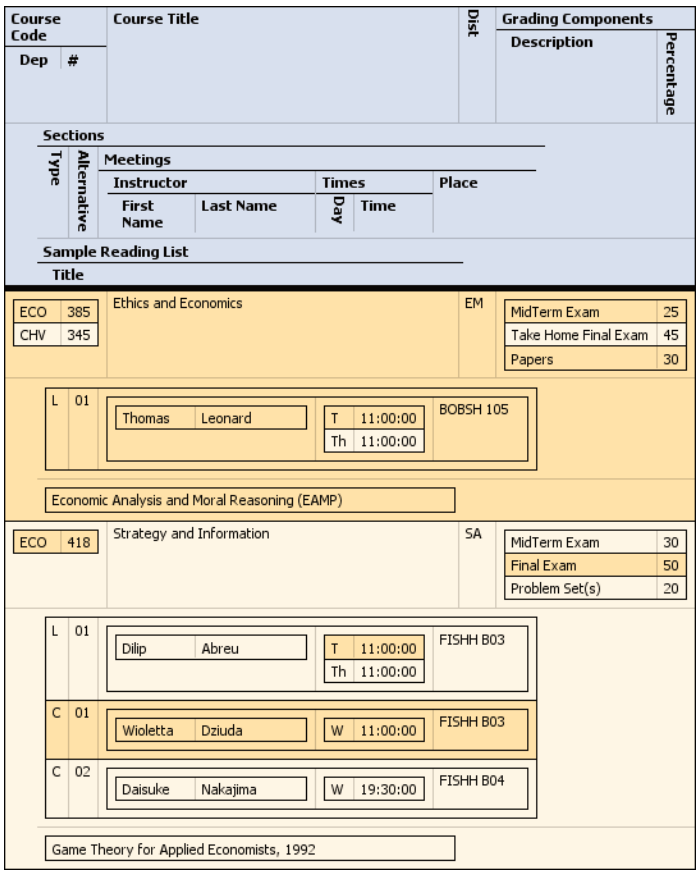


Figure 5-4: An older version of the layout system, exhibiting various readability problems due to (1) uncollapsed borders around every relation value, (2) alternating row colors at more than one relation level, and (3) wrapping of columns within table rows.

Nested relational data is most commonly illustrated in a *nested table* layout. An annotated example of such a layout is shown in Figure 5-3; it renders data about the first few courses in an academic course catalog. A nested table layout consists of a *header* area (shown in blue) and a *content* area (shown in beige). The header area presents the schema structure of the displayed nested relation, with simple labels for columns containing primitive fields and recursive labels for columns containing relation fields. Columns containing relation fields may recursively contain columns for either primitive or relation subfields. In the content area, tuples in the rendered relation are stacked vertically with row separators between them. Each tuple renders its primitive values as simple strings and its relation values recursively as the content area of another nested table.

Since we use nested tables as a base case for our layout algorithm, we experimented with various visual tweaks to make them readable for a wide range of inputs. To minimize visual noise, it is important to insert horizontal borders only *between* tuples rather than on all four sides of them. This also ensures that a relation value containing only a single tuple with primitive fields takes up no more vertical space than the same primitive fields had they not been enclosed in a subrelation. If borders and margins are not carefully collapsed where possible, the result easily becomes a cluttered mess of nested boxes, as illustrated in

Figure 5-4.

While remaining frugal with the use of borders, we found it crucial to leave a small margin to the left and right of each nested relation value, preventing horizontal tuple separators from extending all the way to the vertical column separators. This can be seen in Figure 5-3 for instance in the INSTRUCTORS column. Otherwise, unrelated tuples in sibling relation values may appear to be connected if they line up by accident. During interactive use, the same margin also prevents the cell selection cursor and any column selection highlights from extending all the way to the vertical column separator, providing a further hint to the structure of the data.

Finally, we experimented with alternating row colors to enhance readability. We found this to be effective only if applied solely to the root-level relation. Alternating row colors at more than one level quickly gets confusing, as also illustrated in Figure 5-4.

5.2.2 Outline Layouts

Nested table layouts, like the one previously seen, can quickly become extremely wide if many fields are to be displayed. For on-screen interfaces, the need for horizontal scrolling is undesirable, and for printing on paper, pagination in the horizontal direction is awkward. As another basic alternative to the nested table layout, we consider the *outline* layout, which is common among XML editing tools. Figure 5-1(a) shows the same data as before rendered using such a layout.

In an outline layout, tuples are stacked vertically in an indented bullet list fashion; we show one bullet per tuple. Unlike the nested table layout, which arranges tuples vertically and the fields within each tuple horizontally, the outline layout arranges both tuples and the fields within each tuple vertically. Thus, having more fields in a schema makes an outline layout taller, not wider. In a tuple in an outline layout, primitive fields are rendered as strings with their respective schema labels to the left, whereas relation fields are rendered recursively with their schema labels above, extended to the full width of the layout area. The color scheme is the same (blue and beige) for labels and values in an outline layout as for the header and content area of nested tables, respectively.

The outline layout, unlike the nested table layout, supports the concept of a horizontally constrained width. Like the layouts produced by our final algorithm, an outline layout can be produced for any available width (with some minimum constraints), breaking text in primitive value areas as necessary. Our system distinguishes between fixed- and variable-length primitive fields; value areas for fixed-length primitive fields are always rendered at their predicted width, whereas areas for variable-length primitive fields are rendered to the full available width. This ensures that the visual width of a particular primitive field remains the same between successive tuples, even when the actual value differs. The rationale is two-fold. First, if layouts are to be used for data entry or data editing in a database application, the width of an input field should be commensurate with the expected size of its values. Second, keeping the perceived overall shape (*gestalt*) of successive tuples of the same relation schema similar should make it easier for a user to visually scan for specific fields in those tuples.

5.2.3 Hybrid Layout

Whereas nested table layouts quickly grow wide, outline layouts tend to be tall and narrow. Outline layouts use space inefficiently by only starting values on the left-hand side of the page, and by repeating schema labels once for every value. They are also harder to read than table layouts, since they tend to put values from the same schema field but from different tuples far apart.

In Figure 5-1(b), we introduce a hybrid layout that embeds nested tables inside an outline layout. This saves vertical space compared to the pure outline layout in Figure 5-1(a). Like for the outline layout, we assume that we are given a constrained amount of horizontal space to work with, such as the screen size or page width for printing. We then create a layout that is guaranteed to fit within this horizontal space. This avoids horizontal scrolling or pagination, and ensures that the layout will only grow in the vertical direction as more tuples or fields are added.

When building a hybrid outline/table layout, the algorithm must start at the root level of the relation schema and decide, for each relation field, whether to render that relation field using a nested table or another level of an outline layout. If the decision is made to render a relation field using an outline layout, the decision process is repeated recursively for each of its fields. If the decision is made to render a relation field using a nested table layout, all child fields are rendered using a nested table layout as well. While our recursive layout generation algorithm technically supports embedding outline sublayouts into nested table layouts, this makes little typographical sense, and we do not make use of this case.

The decision to use an outline layout vs. a nested table layout for a given relation schema could reasonably be made using a cost optimization strategy, for instance based on the total area consumed by the layout in each case. However, because nested table layouts almost invariably consume less area than corresponding outline layouts, a simpler heuristic is possible: always use a nested table layout if there is enough horizontal space available for it. This is the rule used in our algorithm; it is illustrated by contrasting the narrow layout in Figure 5-1(b) with the wider layout of Figure 5-1(c). In Figure 5-1(b), the layout is constrained to a small width, and small nested tables have been chosen by the heuristic for relation fields Course Listings, Grading, Sections/Meetings, and Sections/Instructors. All higher-level relations are rendered using outline sublayouts. In Figure 5-1(c), the same data is rendered at a larger constrained width, allowing both the Sample Reading List relation field and the entire Sections relation field to be rendered as a nested table. Note that a hybrid layout that is given enough horizontal space to work with will always degenerate to a pure nested table layout.

Since the outline vs. nested table decision heuristic depends on whether or not there is enough horizontal space for a table, the determination of minimum table column widths is important at this stage. For relation fields, table columns are as wide as the sum of their child fields' columns, plus separator lines and side margins, as well as any extra space needed to accommodate the relation column's own header label. For fixed-width primitive fields, the width of the table column is simply the width of the field. For variable-width primitive fields, we experimented with various heuristics, and found the average width of values in the field to be a sensible minimum, limited upwards to a constant value. The latter constant should be within the recommended width of a standard book column, e.g. on the

order of 50 characters[1]. Primitive columns also need to be wide enough to accommodate their schema labels, which may often be wider than the actual values in the fields. For the latter case, we automatically use vertical column labels in tables if this makes the column narrower for the purposes of the outline vs. nested table decision. Examples are the FORMAT and NUMBER column header labels in Figure 5-1(c).

After the minimum widths of table columns have been determined and the decisions to use outline vs. nested table sublayouts at each relation level have been made, additional horizontal space may be available to the right of nested table sublayouts. A separate *table justification* step uses the remaining horizontal space to first, for readability, make any previously vertical column headers' labels horizontal. This is done in a greedy order to minimize the number of remaining vertical column labels. Then, any remaining horizontal space is distributed among columns holding variable-length primitive fields, in proportion to their fields' average lengths. The table justification step is illustrated in the transition from Figure 5-1(c) to Figure 5-1(d).

Note again that layout styling decisions, such as table column widths or whether to use an outline or a table sublayout for a given subrelation, are made once for each field in the schema of the input data rather than once for each value in the input data. This means, for instance, that in a given layout like Figure 5-1(c), every instance of the SECTIONS relation will be rendered in the same way (either as a table or an outline), regardless of its actual content in each instance. The rationale is similar to that for making primitive fields in outline mode always the same width. Note, however, that while the size and position of fields in a layout will always stay consistent in the horizontal direction, individual text boxes and lists of tuples may grow and shrink in the vertical direction, depending on the data that is being laid out.

5.2.4 Columns in Outline Layouts

While the hybrid layouts layout shown in Figures 5-1(b), (c), and (d) save substantial area compared to corresponding outline layouts, they still use horizontal space inefficiently in cases where small primitive fields can not be made part of a table and where only tables with narrow content can be used. In form-style database user interfaces, the traditional solution is to make use of multiple columns of fields. Note that these are a different kind of columns than the columns in nested tables; they allow different fields in an outline layout to be organized in multiple adjacent stacks. We now show how our system can automatically incorporate columns in outline sublayouts with no manual styling required.

We considered various approaches to the problem of introducing columns into outline layouts. Design questions include how to pick the right number of columns to use, how to pick the width of each column, whether to allow certain fields to span multiple columns, and after which fields a new column should be started. We decided to make two simplifying assumptions which seem to work reasonably in practice: (1) the number of columns to use is based solely on the available horizontal space, and (2) every adjacent column has the same width. So for a layout of a width corresponding to a typical letter-size page, for

[1]Robert Bringhurst's *The Elements of Typographic Style* recommends 40 to 50 characters for multi-column text; see http://webtypography.net/2.1.2.

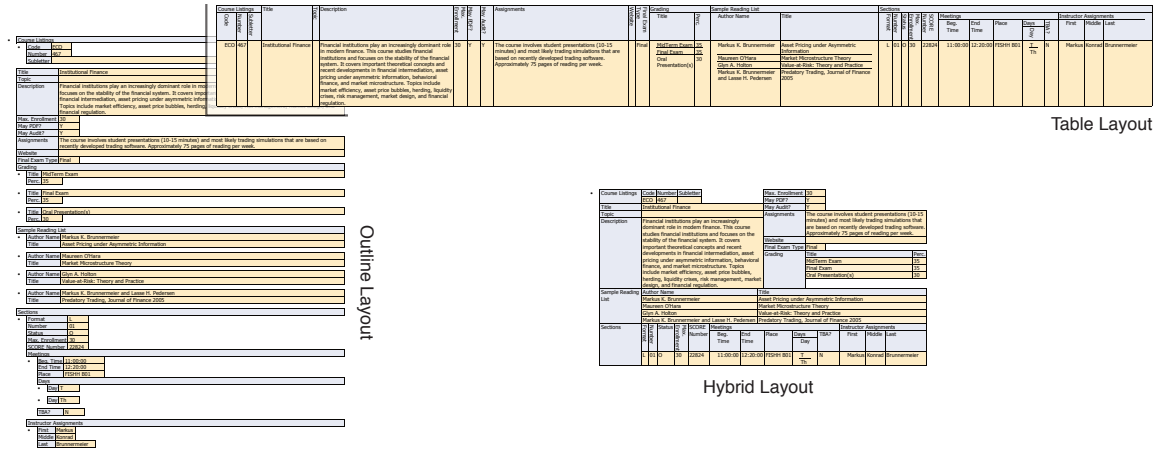


Figure 5-5: A comparison between our hybrid outline/table layout and a pure nested table layout and a pure outline layout, for the case of displaying a single tuple with many fields, including relational fields containing other nested tuples. Each layout is showing the same data in its entirety, at the same scale and font size. Outline layouts waste space by concentrating data to the left of the screen and by repeating labels for each value. Table layouts waste space when different fields in the same subrelation require different amounts of vertical space. Table layouts also tend to become very wide, requiring horizontal scrolling if viewed on a screen.

instance, the algorithm would use a two-column layout in the root level outline. However, one complicating issue must still be dealt with: relation fields that could be rendered in a table layout need to be allowed to span multiple columns if necessary. We settled on the following rule: any relation field in an outline is excluded from participating in a set of multiple columns if that would cause it to be rendered as an outline. There is no requirement that the field would actually have to be rendered as a nested table if excluded, but if the excluded field is rendered as an outline, that outline layout is subject to the usual heuristics about whether to use columns or not at that next level.

For implementation purposes, the algorithm divides the fields of a relation displayed in an outline layout into multiple *column sets*, each which contains again a list of *columns*, each which contains the fields in the column. To populate column sets, the algorithm iterates over outline fields in the order they appear in the schema, assigning each to the current column set. If an excluded field is encountered, it is assigned to a new column set of its own, and a new current column set started. No reordering of fields is done at any point, as the order of fields in the schema is considered significant for presentation purposes. After all fields have been assigned to column sets, each column set partitions its assigned fields into columns. Figure 5-1(e) shows the final hybrid layout with column support. The example layout has a single outline sublayout with 3 column sets; the first column set has two columns and contains the fields COURSE LISTINGS through DESCRIPTION in the first column and the fields MAX. ENROLLMENT through GRADING in the second column. The second and third column sets each have only one column with a single field in each, containing the fields SAMPLE READING LIST and SECTIONS, respectively.

The partitioning of columns in a column set, that is, after what fields to start each

column, requires a heuristic. Simply giving each column the same number of fields does not work well, since some fields frequently take up more vertical space than others. A better approach is to split the columns so as to minimize the total vertical space consumed; this can be done easily with a dynamic programming routine. However, if this is done independently for each tuple being rendered, two tuples might end up having differently partitioned columns, with different fields starting the columns in each case. This is not ideal for readability. Instead, as before, we make the decision of where to begin new columns only once for the entire layout.

To allow calculation of optimal column partitioning positions on the basis of only aggregate information about the input, we allow our layout generation algorithm to generate *schema-only* versions of sublayouts. For instance, the schema-only layout for the layout in Figure 5-1(e) is shown in Figure 5-1(f). In a schema-only layout, variable-length primitive fields are sized, with simulated line breaking, according to the average width of the field in the entire dataset. Relation fields are rendered, in outline or table form, with a single placeholder tuple only, but subsequently padded such that the size of the subrelation layout is proportional to the average cardinality of that relation throughout the entire dataset. Thus in Figure 5-1(f), the DESCRIPTION field is taller than the TITLE field, and there are about two rows' worth of vertical space allocated in the SAMPLE READING LIST table. Optimal column partitioning decisions are then made using these estimated schema-only layouts. The final layout is shown in Figure 5-1(e) (a larger version was seen in Chapter 1, Figure 1-5).

The final class of hybrid layouts produced is considerably more compact than both the outline and the nested table layouts, and can be produced automatically with no manual input. Figure 5-5 shows a scale comparison between the three layout styles, each showing a single nested tuple from the course catalog example. In this case, the hybrid layout would permit significantly more data to be fit on a single screen without scrolling in either the vertical or the horizontal direction. We can also see that large nested table layouts often waste space whenever two values in the same tuple take up different amounts of vertical space.

5.2.5 Implementation

To produce the hybrid layouts described, our system makes two passes over the input data while maintaining a *stylesheet* as the only other common data structure. Like the SIEUFERD query model, which was described in the previous chapter, the stylesheet maps schema fields and property types to property values, that is, each field in the schema has one value for each property. See Table 5.1 for the list of stylesheet properties. Before the algorithm starts, a subset of properties will already have been set as constants, such as the choice of fonts and separator styles. Our algorithm's first pass over the input data is during the MEASURE phase, which finds the average rendered width of each primitive value and the average cardinality of each relation value. The application of heuristics to set remaining stylesheet properties is then done in a subsequent pass over the schema only, the AUTO-STYLE phase. Finally, the output layout is constructed in the LAYOUT phase, which is the second pass over the input value. The AUTO-STYLE and the LAYOUT phases execute the same code, but with the AUTO-STYLE phase traversing a schema-only version of the data structure used to maintain context, and using the aforementioned heuristics to set undefined

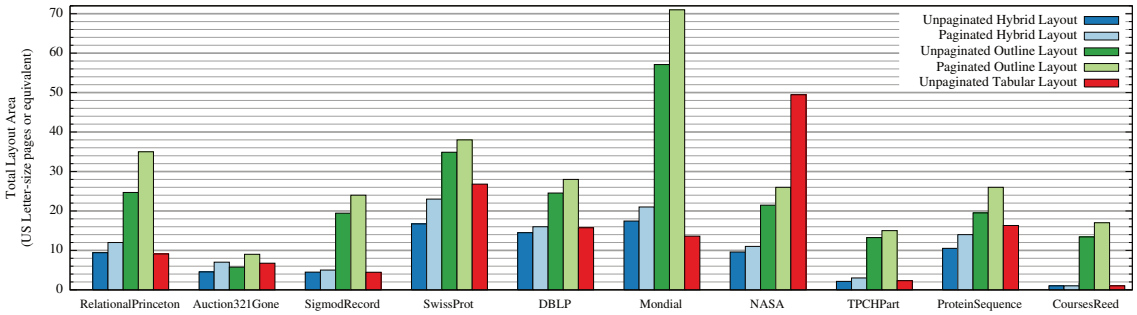


Figure 5-6: Total area consumed by layouts of each of the three types. Outline and Hybrid layout widths are constrained to 8 inches, and are shown both with and without pagination.

stylesheet properties whenever they are encountered. The heuristics for setting stylesheet properties during the LAYOUT phase are described in Table 5.1.

5.3 Evaluation

We evaluated three aspects of our system: runtime performance, the area consumed by generated layouts, and the readability of large layouts as measured by the time taken for human subjects to solve question tasks about the rendered data. As sample datasets, we picked one XML file from each of the 10 categories in the XML Data Repository at the University of Washington[2], except for the Treebank dataset, which is the only one with a recursive schema. We used the "preview" version of each dataset to make sure visualizations would be of a realistic size for human perusal. We also included one dataset from a relational database containing the complete course catalog for a semester at Princeton University (RELATIONALPRINCETONHUGE), and a subset containing only the courses from one department (RELATIONALPRINCETON). See Table 5.2.

5.3.1 Runtime Performance

For runtime measurements, we ran all phases of the layout algorithm in sequence, and repeated the entire sequence multiple times. The runtimes for individual phases were averaged, less initial dry runs. Resulting runtime statistics for two datasets are shown in Table 5.3, for 30 runs plus 3 dry runs. The machine used had an Intel Core 2 Duo CPU and 4GB of RAM.

Our two sets of runtime measurements suggest, as expected, that both the MEASURE and LAYOUT phases run in time roughly proportional to the size of the input data, as measured by the size of the output layouts. Also as expected, the time consumed by the AUTO-STYLE phase does not depend on the size of the input data, as it depends only on the schema and input stylesheet. For the larger RELATIONALPRINCETONHUGE dataset, the fact that the LAYOUT phase does not take significantly more time to run than than the MEASURE phase suggests that the main bottleneck of the LAYOUT phase is the line breaking code that determines

[2]G. Miklau, http://www.cs.washington.edu/research/xmldatasets/www/repository.html

Table 5.1: Properties defined, for each field in the schema, by a stylesheet. P, R, and P R indicate properties applicable to primitive fields, relation fields, or both, respectively. We have omitted color- and border-related properties.

	Basic Styling Constants
R	**OutlineBulletStyle.** Bullet type for bulleted tuples in outline sublayouts.
R	**OutlineIndentWidth.** Indentation amount for bulleted tuples in outline sublayouts.
R	**OutlineTupleSpaceHeight.** Vertical space between tuples in outline sublayouts.
R	**TableNestSpaceSideWidth.** Horizontal margin amount for nested tables.
P R	**LabelTextStyle.** Text style for outline labels or table headers.
P	**ValueTextStyle.** Text style for values in outline or table sublayouts.
	Constant Heuristic Parameters
R	**OutlineMaxLabelWidth.** Maximum width of labels in outline sublayouts.
R	**OutlineColumnMinWidth.** Minimum width of each outline column.
P	**OutlineMinValueWidth.** Minimum width of a primitive value in an outline.
P	**OutlineSnapValueWidth.** Multiple to round up to when setting the width of a non-variable primitive value in an outline sublayout.
P	**TableMaxPrimitiveWidth.** Maximum width that can be allocated to a table sublayout column with variable-length primitive values, before table justification.
	Properties Set During Measure Phase
R	**OutlineLabelWidth.** Width of labels in outline sublayouts. Sibling fields all use the same width, which is defined at the parent relation level.
R	**AverageCardinality.** The average number of tuples in each subrelation.
P	**IsVariableLength.** Whether a primitive field holds long strings of variable length.
P	**ValueDefaultWidth.** Average width of primitive values in this field when rendered with ValueTextStyle, or maximum width for non-variable length fields.
	Properties Set During Auto-Style Phase
P R	**StartNewOutlineColumn.** True for the first field in each column of an outline column set. The heuristic partitions columns based on schema-only layouts that use ValueDefaultWidth and AverageCardinality to estimate field sizes.
P R	**StartNewOutlineColumnSet.** True for the first field in each outline column set. Column sets allow some sibling fields to be organized in multiple columns and others not. The heuristic puts a field in its own single-column column set if it would otherwise contain an outline sublayout.
P R	**UseVerticalTableHeader.** Whether to display a label in a table header vertically. The heuristic initially assumes vertical labels for primitive fields if this makes the column narrower, but restores as many horizontal labels as possible when the table is justified.
P R	**TableColumnWidth.** The width of each table column. Primitive columns are ValueDefaultWidth wide before justification; relation columns are the sum of their children plus twice TableNestSpaceSideWidth. Columns are also extended to accomodate their labels, as necessary.
R	**UseTable.** Whether to use an outline or a table sublayout for this relation. A table sublayout is used iff it its width before justification is less than or equal to the available horizontal space.

Table 5.2: Quantitative statistics related to the size and complexity of datasets referred to in this chapter. The *depth* of a primitive value is the number of enclosing relation values that must be traversed to reach the primitive value from the root. The *plural depth* only counts non-singleton enclosing relations.

Dataset	*# Characters in Primitives*	*# Primitives*	*# Tuples*	*Mean Depth*	*Max. Depth*	*Mean Plural Depth*	*Max. Plural Depth*
RelationalPrincetonHuge	828 463	59 887	18 751	2.25	4	2.25	4
RelationalPrinceton	21 811	1 359	419	2.20	4	2.20	4
Auction321Gone	13 679	250	61	2.96	4	1.00	1
SigmodRecord	12 360	953	560	5.01	6	2.51	3
SwissProt	13 559	1 566	809	3.15	5	1.94	3
DBLP	22 012	1 493	310	2.10	3	1.09	2
Mondial	15 046	3 032	1 020	3.63	5	2.48	4
NASA	18 218	854	731	6.16	8	3.43	5
TPCHPart	10 607	901	101	2.00	2	1.00	1
ProteinSequence	11 236	953	494	3.85	6	1.94	3
CoursesReed	4 858	800	200	2.33	3	1.00	1

Table 5.3: Runtime measurements for each phase of the layout algorithm. Standard error is within 3% in each case.

Dataset	*Pages*	*Algorithm Phase Runtime (s)*			
		Measure	*Auto-Style*	*Layout*	*Paginate*
RelationalPrinceton	12	1.03	0.29	0.83	0.0014
RelationalPrincetonHuge	455	40.16	0.31	42.43	0.0451

the size of rectangles assigned to display primitive values, since said line breaking code is common to both phases. Profiling has confirmed this to be the case. For the smaller RelationalPrinceton dataset, the time to perform Auto-Style and Layout for a new width is interactive. Pagination time is insignificant, but roughly proportional to the number of pages in the output.

5.3.2 Layout Space Efficiency

To evaluate the space efficiency of our layouts, we compared, for each dataset, the area consumed by our own hybrid layout vs. the area consumed by a pure outline layout and a pure nested tabular layout. Each layout was produced by our layout manager, with the latter two using a hard-coded value of false and true for the UseTable stylesheet property at every field to force the layout manager into pure outline and pure nested tabular mode, respectively.

Figure 5-6 compares the total area of each kind of layout for every dataset. The unpaginated area of a layout is that of the smallest rectangle enclosing it. The paginated area, for hybrid and outline layouts, is the number of pages consumed by the layout times the imageable (non-margin) area available on each page. Thus, the latter includes space wasted at the end of each page whenever the pagination algorithm has opted to break the page at an earlier but less awkward place. Since pure tabular views of an entire dataset are generally too wide to fit on a regular letter-size page, the tabular layout is rendered as a single, very large page. In cases where the tabular layout is actually narrow enough to fit on a page, notably the the SigmodRecord, TPCHPart, and CoursesReed datasets, the hybrid layout is nearly identical to the tabular layout, except that the pagination algorithm may be used to break up the hybrid layouts in the vertical direction. The ratio of the area consumed by an unpaginated outline layout to that consumed by an unpaginated hybrid layout is 3.9:1 on average. Similarly for tabular to hybrid layouts, it is 1.6:1 on average.

Looking at the data from Figure 5-6, we see that the hybrid layout consumes less area than the corresponding outline layout for every dataset, with or without pagination enabled. The difference is greatest, between 4 and 13 times, in the cases where the hybrid layout corresponds to a pure nested tabular layout, namely SigmodRecord, TPCHPart, and CoursesReed. In these cases the schema of the data was flat or almost flat, and so a standard table layout would make very efficient use of the space. In the other cases, the hybrid layouts are about half the size of outline layouts on average. The smallest difference was for the Auction321Gone dataset, where the outline layout was 1.3 times the size of the hybrid layout. In this case the schema was nested in several levels, but contained only singular relations (relations only ever holding a single tuple) beyond the top level, so there were no opportunities for the hybrid layout algorithm to introduce tables into the lower levels of the layout. The modest saving over the outline layout came from the hybrid layout's ability to display data in an outline tuple over two columns. A more significant difference was for the Mondial dataset, where the outline layout was 3.3 times the size of the hybrid layout. Here the hybrid layout made good use of both tuple columns and the ability to render subrelations as tables, and only used one level of outline bullets.

While pure nested tabular layouts cannot be constrained to a page width like the outline and hybrid layouts, they tend to consume less total area than the outline layouts. See

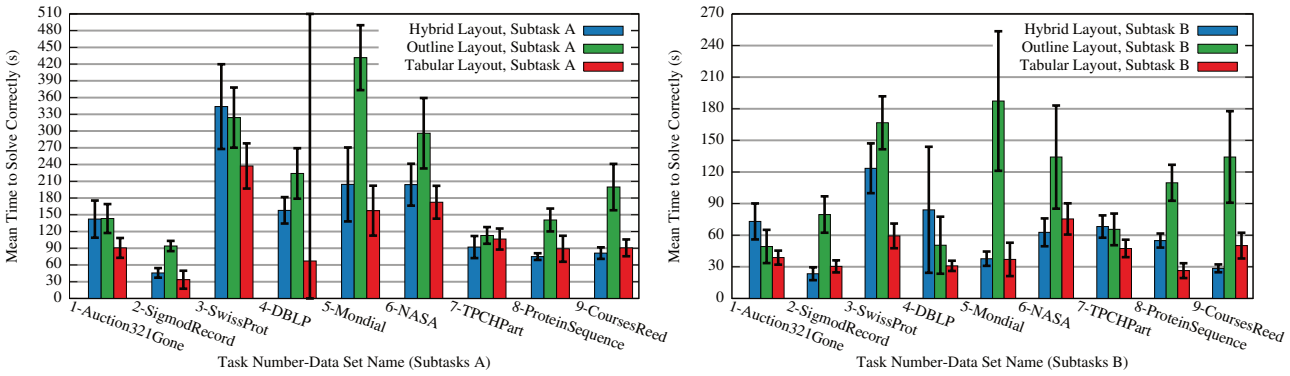


Figure 5-7: The mean time to solve each task in the user study, grouped by the kind of layouts that were used to generate the PDFs subjects used to solve the task. The error bars show the Standard Error of the Mean. Subtasks B were given to subjects directly after Subtasks A, and were in each case identical to Subtask A except for a small emphasized change in the question text.

again Figure 5-6. For AUCTION321GONE, NASA, PROTEINSEQUENCE, SWISSPROT, the hybrid layout still consumes between 30% and 80% less area than the tabular layout. This is because nested tables waste large amounts of space whenever a row contains cells of variable heights, such as when one empty and one well-populated subtable are placed horizontally adjacent to each other on the same row of a parent table.

5.3.3 Readability

To evaluate the readability of our layouts, we conducted a between-subjects online user study using Amazon Mechanical Turk[3] and StudyCaster, a Java-based tool we developed to allow test subjects to stream timestamped recordings of their computer screens to our server with a minimum of effort. In an initial public recruiting stage of the study, workers were offered $0.25 to launch the StudyCaster and solve a chart-making task that required the workers to have Microsoft Excel installed on their machines. In the second and main stage of our experiment, we gradually invited qualified Mechanical Turk workers from the first stage directly to do a second task, worth $3.00. This task contained, for each subject, 9 different two-part questions, each two-part question being based on a separate PDF file with a layout generated from one of our 9 XML sources from the UW XML repository. The questions were a mix requiring the subjects to do both scanning across multiple similar entities (e.g. “What is the Brand number of the product sold in a Jumbo Bag container at a Retail price of less than $950?” or “How many articles were published in Volume 12, Number 3?”) and lookup between the attributes and related entities of a single entity (e.g. “What is the name of the person who was responsible for digitizing the earlier work by authors X?”). See Figure 5.4. To reduce potential noise from subjects’ varying familiarity with their PDF readers’ search feature, the PDF files were rasterized, effectively disabling the feature for everyone. All subjects were given the same questions

[3] http://www.mturk.com

#	*Data Set*	*Question Given (subtask A/B variations in curly brackets)*
1	Auction321Gone	What is the size of the hard drive of the computer with the {*shortest*|*longest*} "Time Left" on the auction?
2	SigmodRecord	How many articles were published in {*Volume 12, Number 3*|*Volume 14, Number 1*}?
3	SwissProt	What organism ("Species") has an author with the last name "{*Poovaiah*|*Cognetti*}" listed among its references?
4	DBLP	How many articles are listed under the "{*Inproceedings*|*Mastersthesis*}" category?
5	Mondial	What percentage of the population of {*Germany*|*Gibraltar*} is listed with "Roman Catholic" as their religion?
6	NASA	What is the name of the person who was responsible for digitalizing ("Ingesting") the earlier work by authors {*Spencer and Jackson*|*Larink, Bohrmann, Kox, Groeneveld, and Klauder*}?
7	TPCHPart	What is the "Brand" number of the product sold in a "Jumbo Bag" container at a "Retail-price" of {*less*|*more*} than $950?
8	ProteinSequence	What is the last name of the first author listed in the references for the protein identified by the "Id" code "{*CCMQR*|*CCWHC*}"?
9	CoursesReed	How many 0.5-unit {*Chemistry ("CHEM")*|*Biology ("BIOL")*} courses are mentioned in the list?

Table 5.4: Question tasks given in the user study.

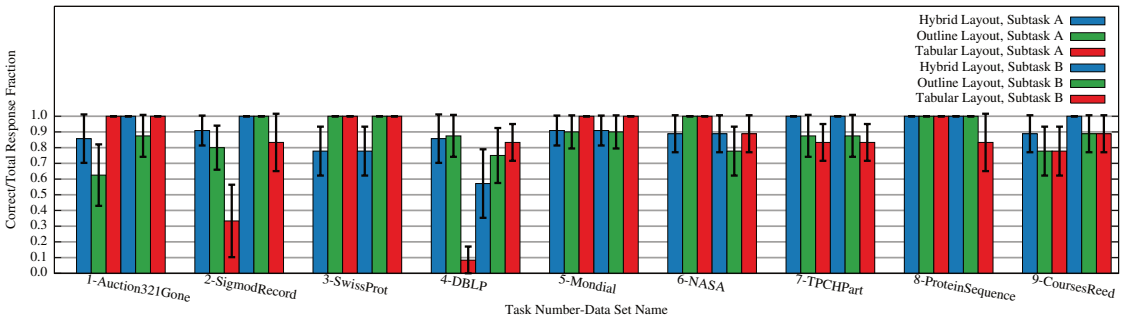


Figure 5-8: The fraction of correct responses to each task in the user study. The error bars show the Standard Error of the Mean when assigning value a value of 1 for correct answers and 0 for incorrect answers.

and datasets in the same order. However, the type of layout provided for each dataset was randomized, with the constraint that each subject would see 3 datasets rendered with each of the 3 kinds of layout types. The order of the layout types was round-robined such that datasets number 1, 4, and 7 would use the same layout types, as would 2, 5, and 8, and as would 3, 6, and 9. Each of the 18 total questions (from 9 two-part questions) would be shown in the StudyCaster pop-up window, which allowed us to measure the exact amount of time the subjects spent viewing, and hence presumably spent working on, each question. The StudyCaster software also allowed us to further limit timings to when workers had the correct PDF file in focus in their PDF reader to answer the currently shown question (sampled from the Win32 API at 5Hz), and to exclude time idle more than 5 seconds from keyboard or mouse activity (same). The idle time rule was used to decrease noise from workers taking a break from the computer while having a question open on the screen.

Our user study yielded data from 27 subjects. An additional 6 subjects completed the study, but were not included in the dataset due to technical problems uploading their screen recordings and timing data. The 18 task questions were answered correctly by 88% of subjects on average, with too limited variation to draw conclusions about possible impact

Table 5.5: Summary of statistical tests run on the dataset from Figure 5-7. Only tasks for which the ANOVA yielded $p<0.05$ are shown. For Tukey HSD pairs with $p<0.05$, we also show the relative differences in average task completion times.

Task	*Levene* (hetero-scedasti-city)	*ANOVA*	*Tukey HSD* (follow-up to find differences between pairs)			*Mean-Time-to-Solve Ratio*	
			Outline v. Hybrid	Outline v. Tabular	Tabular v. Hybrid	Outline: Hybrid	Outline: Tabular
#	*p*	*p*	*p*	*p*	*p*		
2A	0.5312	0.0012	**0.0019**	**0.0169**	0.8006	2.05	2.80
2B	0.0627	0.0034	**0.0036**	**0.0448**	0.9279	3.40	2.62
3B	0.2889	0.0032	0.3047	**0.0023**	0.0962		2.81
5A	0.7976	0.0072	**0.0200**	**0.0142**	0.8521	2.11	2.74
5B	0.0389	0.0174	**0.0251**	0.0532	0.9999	4.97	
8A	0.1099	0.0101	**0.0092**	0.0996	0.8201	1.88	
8B	0.3409	0.0004	**0.0048**	**0.0008**	0.3129	2.00	4.16
9A	0.4104	0.0034	**0.0050**	**0.0119**	0.9553	2.46	2.20
9B	0.1812	0.0107	**0.0108**	0.0530	0.7919	4.69	

of layout type on correctness. See Figure 5-8. Figure 5-7 shows the average time taken to complete each subtask for each layout type. Each subject's timing is included in the average for each subtask only if the subject answered that question correctly.

To test our user study for statistical significance, we considered the timing data from each of the 18 subtasks separately. In each case, we thus had 3 sets of measurements of the time taken to solve the task correctly, one for each kind of layout presented to the user. Timings for incorrectly solved tasks were omitted for this part of the data analysis. We first ran Levene's test to confirm that our experiment design conformed to ANOVA's assumption of homogeneous variances between the 3 measurement populations in each case. In only 1 of the 18 cases (Task 5B) was Levene's test significant (indicating non-homogeneous variances) with $p < 0.05$, suggesting that this is a reasonable assumption. We thus proceeded to use ANOVA to analyze the results, with a Tukey Honest Significance Difference (HSD) test as a follow-up in cases where the ANOVA was significant. Since we are doing 18 tests, we should require $p < 0.05/18$ for strictly significant ANOVAs, as per the Bonferroni correction. There are two significant results to this confidence level, for tasks 2A and 8B. For the purposes of discussing results, however, we have done the Tukey HSD follow-up test for all tasks with ANOVAs up to $p < 0.05$. This allows us to list all the most significantly different pairs of timings between different layout types, as shown together with the relevant p-values in Table 5.5. Note that we can expect about one $(18 * 0.05)$ of the borderline-significant ANOVAs in this table to be due to chance.

Looking at the follow-up tests from Table 5.5, we can see no significant differences in the task completion times between the tabular vs. the hybrid layouts. We do however see consistent differences both between the outline and the hybrid layouts as well as between the outline and the tabular layouts. These differences are present in both subtasks of several questions, suggesting that they are relevant both when users are first learning to do a task and when they immediately after do a second similarly structured task. In terms of relative

task completion times, it is clear that both the hybrid and the tabular layouts outperform the outline layout, in both cases being completed 2.9 times faster on average for the tasks listed in Table 5.5. We believe outline layouts are harder to read because (1) they are large and require the user to scroll more to look through a given amount of data and (2) they exhibit less spatial regularity than the other layouts.

The relatively high performance of the nested tabular layouts suggests that for the kinds of large datasets we had our users work with, the very regular structure of the tabular layout can outweigh its disadvantages of taking up more space and requiring both horizontal and vertical scrolling. However, the tabular layout would be a poor choice for smaller datasets, such as the common database application requirement of showing the details of a single entity with all its attributes and related subentities. In this case, data that would fit on a single screenful when formatted with the hybrid layout would likely exceed the width of the screen when formatted with the pure table layout, requiring horizontal scrolling rather than no scrolling at all, and making very poor use of vertical space, since the top-level table would have only a single row. The pure outline layout, on the other hand, would fit in the required horizontal space, but would likely exceed available space in the vertical direction, also requiring scrolling. Future evaluation could focus on layout performance on smaller, form-size datasets.

In terms of space efficiency, outline views lose in most of the cases, with hybrid layouts performing significantly better. In almost every case, unpaginated hybrid layouts also perform as well as or better than unpaginated tabular layouts. It is worth noting that tabular layouts are not, by their nature, constrained horizontally. So they can not, for instance, be printed on paper, unlike the hybrid layouts. The user study shows a less clear difference, though outline views to still tend to fare the worst. It is slightly surprising to see the very large pure table layouts outperform the other layouts in a few cases; this is an interesting result.

5.4 Extensions

This section discusses features that were added to the layout generation system after the previously discussed evaluation, in support of the complete visual query system that was discussed in Chapter 3.

5.4.1 Interactive Features

While our evaluation focused on the static aspects of our layout management algorithm, our implemented system now includes multiple features oriented towards interactive use.

A requirement for many kinds of interactions is to be able to make selections among displayed elements in the layout. Our system supports a spreadsheet-like cursor which can be used to select any *cell* in the layout, where a cell is defined as either a label for a primitive field, a label for a relation field, or a primitive value. Selections can be made either by clicking the mouse or by moving the cursor with keyboard navigation keys (arrow keys, HOME/END, and PAGE UP/PAGE DOWN). Because the aforementioned definition of a cell serves to ensure that no two cells can ever overlap, determining the cell to be selected in the

case of a mouse click is a simple matter of determining what cell occupies, or is closest to, the point at which the mouse was clicked. For keyboard-based cursor movement, we found the cursor behavior to feel the most natural when the relative motion of the cursor followed the physical location of the cells in the visual layout rather than the logical location of the cells in the schema. To make keyboard cursor movement work well when traversing cells arranged in various non-trivial configurations, we store the cursor state as an (x,y)-position on the layout rather than simply as a pointer to the selected cell. This generalizes the behavior seen in existing spreadsheets for instance when moving the cursor across merged cells.

By dragging the cell cursor across fields in a layout, or by moving the cell cursor while pressing appropriate keyboard modifiers (SHIFT or CTRL/COMMAND), the user may select any number of fields at a time. When the layout system is used as a front-end to our visual query system, such *multiple selection* can be used with query actions such as HIDE and CLEAR FILTER in order to quickly apply the action to multiple fields. Actions such as JOIN, FILTER, and SORT ASCENDING also define meaningful behaviors on multiple selections. Note that our current system only supports multiple selection of *fields* (e.g. table columns), since all the operations of our visual query language are designed to operate on fields rather than, say, tuples or individual cells. Once our system is extended to support data editing, support for multiple selection of tuples or cells would be useful as well, for instance for use with operations such as DELETE.

An additional feature is the ability to interactively override the stylesheet settings made by the automatic layout manager. This has the potential to significantly improve readability of output layouts, since the user can use their domain knowledge to decide where labels are superfluous and can be omitted, what fields should serve as titles and thus be emphasized with larger fonts, and such.

Finally, our system supports "frozen" table headers which stay put at the top of the screen for as long as a table is partially visible in the scrolling viewport. This works both for pure table layouts and where table layouts are contained within outline layouts (hybrid layouts).

5.4.2 Stable Interactive Layouts

In the standalone version of our layout algorithm, the measurements made in the MEASURE phase are done on the same input data that is subsequently laid out in the LAYOUT phase. This means that two datasets with identical schemas may nevertheless be rendered using different layout decisions. In our visual query system, it is desirable for output layouts to stay consistent across the many intermediate results that are generated as the user performs a sequence of query manipulation actions. For example, the width of table columns should not usually change as a result of applying a filter or sorting operation, as this would make it difficult for the user to see exactly what changed as a result of applying the operation in question. Similarly, the retrieval of more rows during infinite scrolling should retain the layout decisions that were made for the original dataset.

In our visual query system, the properties that make up the layout algorithm's stylesheet (Table 5.1) can be stored alongside the properties that define the SIEUFERD query model (Table 4.1). The combined data structure defines not only the query to be executed to

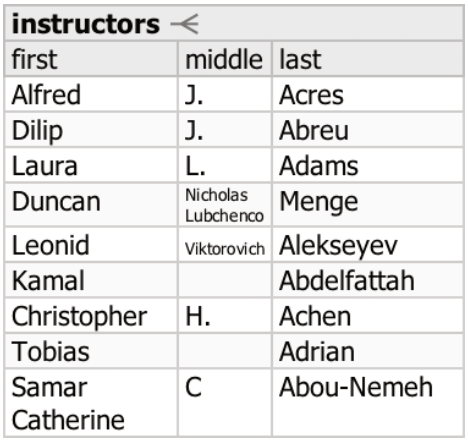


Figure 5-9: Strategies used to render exceptionally long string values at a given prescribed width. Text is broken at word boundaries (SAMAR CATHERINE), or the font size is decreased (VIKTOROVICH), or both (NICHOLAS LUBCHENCO). When sufficient vertical space is available, text rendered at a smaller font size is shifted down to ensure baseline alignment with text in adjacent cells (VIKTOROVICH).

retrieve results, but also how to render those results on the screen. Because the layout stylesheet is now part of the state that defines the current query, we can reuse layout-related state from one intermediate query result to the next.

We define an additional stylesheet property SAMPLESIZE, initially zero, which is used to inform an improved version of our layout algorithm's MEASURE phase. A field's SAMPLESIZE is defined as the number of distinct observed values that went into the statistics currently stored about that field's data (i.e. AVERAGECARDINALITY for relation fields, and ISVARIABLELENGTH and VALUEDEFAULTWIDTH for primitive fields). The improved MEASURE phase performs new measurements for a field only if the old sample size is less than 50 and new sampling would increase the sample size by at least 75%, with the latter requirement waived if the old sample size is less than 15 and the new measurement would increase the sample size. These heuristics were developed by trial and error; the specific parameters were chosen to work well given our default query limit of 100 tuples prior to infinite scrolling. In the visual query system, measured stylesheet properties like SAMPLESIZE, AVERAGECARDINALITY, ISVARIABLELENGTH, and VALUEDEFAULTWIDTH are preserved in the event that a field is temporarily hidden, and are also carried along when a relation is copied from one query to another during an automatic or manual join operation.

Since, in particular, the layout property dealing with column widths (VALUEDEFAULTWIDTH) is no longer guaranteed to be based on measurements of the actual data to be rendered, we need a strategy for dealing with exceptionally long string values. The solution is to apply a combination of line breaking and font sizing; see Figure 5-9. Fields for which ISVARIABLELENGTH is true tend to contain long-form text such as comments or descriptions, and are first broken into paragraph lines on word boundaries. Fields for which ISVARIABLELENGTH is false tend to contain data less suitable for line breaking, such as phone numbers, email addresses, mailing address lines, or proper names. In the latter case, or when line breaking still yields a text layout too wide to fit in the available

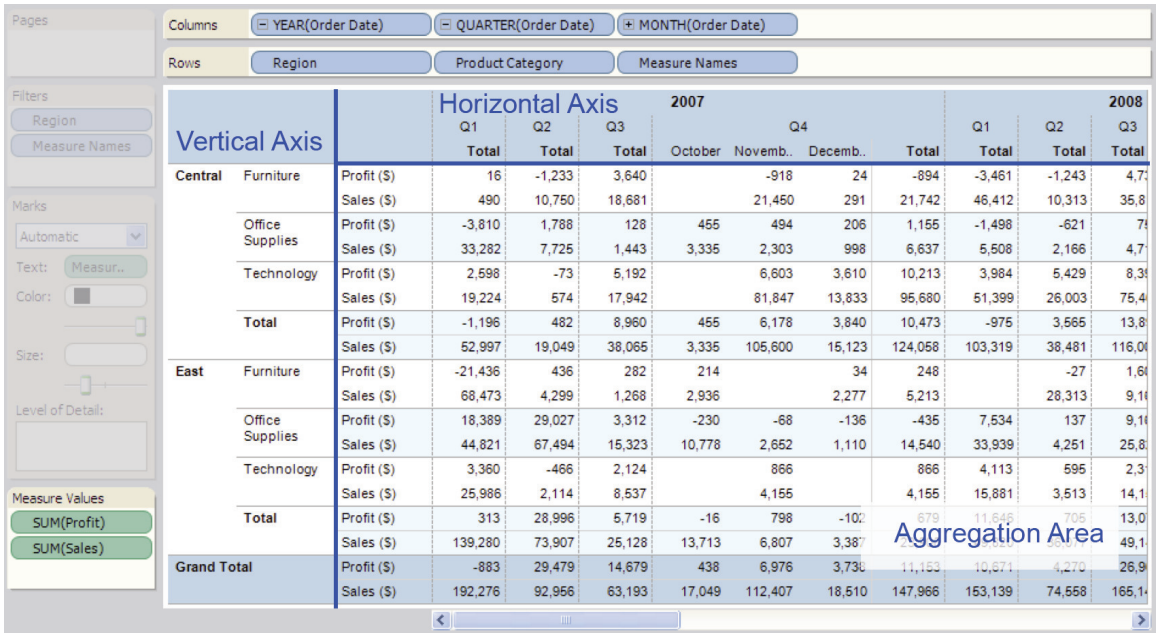


Figure 5-10: An example of a classic crosstab visualization, generated using Tableau (with our annotations). Crosstabs accumulate tuples of data, e.g. (East, Furniture) and (2007, Q4, October), on both the vertical and horizontal axes, and then aggregate remaining fields, e.g. Profit and Sales, in a central area grouped by the intersection of tuples on the two axes.

horizontal space, for instance due to long words, the font size of the entire text layout is decreased by the amount necessary to fit the text layout in the available horizontal space.

A convenient side-effect of stable layouts is that a typical interactive query session spends very little time in the layout's Measure phase; measurements are typically done once and then stay constant. Some extra logic can be added to handle special cases where measurements should always be recalculated, for instance when a formula is changed. Avoiding line breaks in fields for which IsVariableLength is false furthermore speeds up the generation of layouts in the Layout phase, as we can assume certain dimensions for each text box without performing expensive calls into the font rendering subsystem.

5.4.3 Crosstabs

In all the table layouts we have seen so far, the structure of the table header was fully defined by the schema of the input data, with tuples always accumulating under the header in the vertical direction. Another kind of table layout, the *crosstab* (cross-tabulation), works differently. In a crosstab, tuples of data may accumulate on both the horizontal and vertical axes, intersecting to form aggregations in the table's central *aggregation area*. This is the kind of visualization that is produced by Excel's *pivot tables*; see Gray et al. [46] for a discussion. Crosstabs are also supported by most business intelligence tools. Figure 5-10 shows an example, generated by Tableau.

In SIEUFERD, cross-tabulations can be supported as a simple extension to our nested

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

Figure 5-11: Input data for the crosstab example in Figure 5-12, shown here in a regular nested table layout.

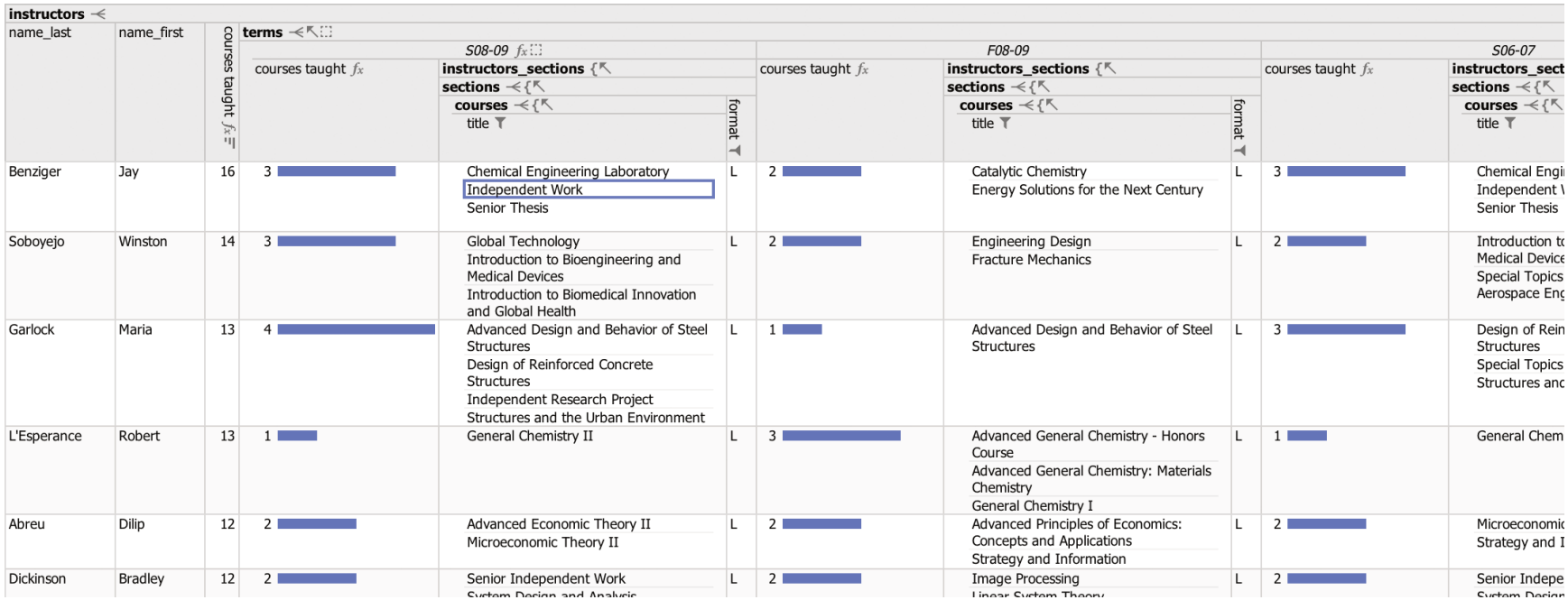


Figure 5-12: A crosstab in SIEUFERD. By allowing data values (e.g. S08-09, italicized) to appear in the table header as grouping keys, tuples can accumulate in both the horizontal (TERMS) and vertical (INSTRUCTORS) directions. The query shown here is identical to that of Figure 5-11, but has a crosstab formatting option enabled on the TERMS relation. All the usual query interface actions remain available from the crosstab layout, including from data values in the table header.

table layouts. In the stylesheet, we add a boolean Crosstab option that can be enabled on any relation field. When enabled, the relation in question has its immediate nested tuples arranged horizontally instead of vertically, with non-aggregate primitive fields automatically projected to form grouping keys in the table header. This is shown in the transition from the plain nested table in Figure 5-11 to a crosstab in Figure 5-12. In the example, for every tuple in the instructors relation, each nested tuple in the terms relation appears in its own nested table column, aligned with corresponding terms tuples in other instructors tuples. Another use case for crosstab layouts is to display key-value data in table form; see Figure 5-13.

Unlike traditional crosstabs, such as those produced by Tableau, SIEUFERD crosstabs work like fully general nested tables. This allows arbitrarily nested data to be displayed in the crosstab's aggregation area. For instance, we can show the exact tuples that contribute to each aggregate value, as shown in the instructors_sections relation in Figure 5-12. Furthermore, all of the SIEUFERD system's query-related actions, such as Filter, Sort, Hide, and editing of labels and formulas, work as usual from within crosstab layouts, including from data values in the table header. The same goes for other interaction features such as cursors, frozen headers, and infinite scrolling, and custom formatting options. One exception is that the Sort action is not meaningful on the crosstab's aggregated fields, e.g. courses taught in Figure 5-12. On the other hand, Sort can be used to define the horizontal order of crosstab tuples, while Filter can be used to cherry-pick them.

In our visual query system, the data tuples that make up a crosstab header are retrieved using a generated SQL query that runs separately from the SQL queries used to evaluate the main result. This generated query is similar to that which would be used to populate the query interface's filter popup if a filter was opened on the Crosstab relation. The retrieved ordered list of header tuples become part of the layout's stylesheet, ensuring that the layout can then be built in a single pass over the main result.

It is not, in general, possible to build the list of crosstab header tuples purely from the main result, since the `LIMIT`-constrained main result is not guaranteed to include every value that should appear in the header, and since the appropriate order of heading tuples would not be inferable. This is the reason why we run a separate SQL query to retrieve crosstab header tuples. The latter separate query can still have a `LIMIT` clause on it, but its `ORDER BY` clause will be configured such that the leftmost N crosstab header tuples are always guaranteed to be retrieved.

5.4.4 Numeric Formatting and Visualization

Is is useful, when generating layouts, to pay some special attention to the formatting of numerical data values. These may originate from the data source itself, or be returned by formula functions such as count and average. Most importantly, numeric values displayed in a table column should always be right-aligned, with the same number of digits (possibly zero) shown after the decimal point for every value in the column. During the Measure phase of our layout management algorithm, we use a heuristic to determine a reasonable number of decimal places to show for numeric values in each primitive field, and then stick to this decision across the entire generated layout. The heuristic is to show the minimum number of decimal places needed to ensure that every observed value is displayed with

OSP_AWARD								
AWARD_EFFECTIVE_DATE	TITLE	Custom Column						
		3K_EQUIP_THRESHOLD		HUMAN_SUBJECTS_FLAG		INTEREST_REVENUE		
		OSP_AWARD_CUSTOM_FIELDS		OSP_AWARD_CUSTOM_FIELDS		OSP_AWARD_CUSTOM_FIELDS		
		value	description	value	description	value	description	
1996-06-01	CAREER: Randomization in Graph Optimization Problems					0	Undetermined or Non Interest Bearing	
1997-09-16	Sloan Research Fellow	Yes	Yes	Unknown	Unkown if this award has human subjects	0	Undetermined or Non Interest Bearing	
1997-10-01	Packard Fellowship in Science and Engineering			Unknown	Unkown if this award has human subjects	0	Undetermined or Non Interest Bearing	
1999-06-15	Merrill Lynch Research-Engineering-LCS	Yes	Yes	Unknown	Unkown if this award has human subjects	0	Undetermined or Non Interest Bearing	
1999-06-15	Randomization and Efficient Algorithms	Yes	Yes	Unknown	Unkown if this award has human subjects	0	Undetermined or Non Interest Bearing	
1999-06-15	Haystack	Yes	Yes	Unknown	Unkown if this award has human subjects	0	Undetermined or Non Interest Bearing	
1999-07-01	Haystack Per-User Information Environment	Yes	Yes	Unknown	Unkown if this award has human subjects	0	Undetermined or Non Interest Bearing	
2000-04-15	Randomization in Graph Optimization	Yes	Yes	Unknown	Unkown if this award has human subjects	0	Undetermined or Non Interest Bearing	
2000-05-01	O2P					0	Undetermined or Non Interest Bearing	

Figure 5-13: Displaying key-value data in a table format using a crosstab layout. The table OSP_AWARD_CUSTOM_FIELDS(AWARD_KEY, FIELD_NAME, FIELD_VALUE) stores data about each OSP_AWARD as key-value pairs instead of keeping each field in its own database column (this schema design antipattern is the "opposite" of that seen in Figure 4-6). A crosstab layout can be used to arrange the fields in table columns for display purposes.

organizations			lobbying by quarter										
spreadsheet	organization	total	2012 q2	2012 q1	2011 q4	2011 q3	2011 q2	2011 q1	2010 q4	2010 q3	2010 q2	201 q1
			amt	amt	amt	amt	amt	amt	amt	amt	amt	
Associations	American Farm Bureau	122,639,168	1,653,052	6,780,455	1,629,825	1,518,799	1,432,736	1,602,048	1,348,055	1,321,743	1,876,591	1,803
Associations	Biotechnology Industry Organization (BIO)	117,651,477	2,701,000	2,581,000	2,595,000	2,685,000	2,615,000	2,545,000	2,970,000	2,815,000	2,735,000	2,930
Plantas	Cargill, Inc.	21,065,239	400,000	440,000	440,000	440,000	460,000	420,000	603,046	555,179	657,042	671
Plantas	Sunoco	19,785,044	450,000	450,000	510,000	580,000	540,000	540,000	520,000	510,000	510,000	510
Associations	National Farmers Union	12,778,592	10,000	20,000	110,000	120,000	80,000	150,000	110,000	200,000	160,000	220
Associations	National Biodiesel Board	11,081,026	499,474	758,948	758,948	454,474	478,987	699,859	471,243	469,672	461,520	431
Associations	Renewable Fuels Association	11,000,939	286,609	315,844	334,502	338,975	598,736	548,500	546,000	263,250	275,161	655
Plantas	Archer Daniels Midland	10,100,000	400,000	390,000	550,000	400,000	700,000	430,000	370,000	330,000	400,000	450
Plantas	Valero Renewable Fuels	10,011,000	215,000	226,000	202,000	219,000	227,000	240,000	220,000	207,000	232,000	235
Associations	Corn Growers Associations	8,928,200	245,000	235,000	175,000	245,000	195,000	205,000	155,000	205,000	155,000	180
Plantas	Murphy Oil	7,440,000	350,000	250,000	230,000	330,000	290,000	200,000	330,000	650,000	680,000	830
Plantas	POET Biorefining	5,951,897	90,000	430,000	210,000	240,000	250,000	460,000	235,000	245,000	255,000	465
Associations	Union of Concerned Scientists	5,925,416	64,371	69,700	96,193	157,867	93,503	167,006	147,577	163,700	458,909	103
Associations	American Chemical Society Green Chemistry Institute	5,719,640	90,000	60,000	50,000	50,000	80,000	80,000	40,000	40,000	50,000	40
Associations	BlueGreen Alliance	5,538,000	200,000	258,000	440,000	265,000	245,000	240,000	1,250,000	810,000	220,000	180
Plantas	Land O' Lakes	5,225,000	360,000	300,000	220,000	300,000	260,000	290,000	230,000	220,000	400,000	170
Associations	Growth Energy	5,104,000	300,000	360,000	400,000	390,000	410,000	370,000	408,000	428,000	466,000	506
Associations	American Soybean Association	3,485,000	40,000	40,000	35,000	30,000	30,000	30,000	30,000	30,000	30,000	40
Associations	Soybean Associations	3,485,000	40,000	40,000	35,000	30,000	30,000	30,000	30,000	30,000	30,000	40
Plantas	Tate & Lyle	2,685,640										
Plantas	Louis Dreyfus Commodities	2,310,050			30,000	30,000	60,000	30,000	30,000	60,000	30,000	60
Associations	American Coalition for Ethanol	1,820,066	111,012	113,812	87,227	91,915	86,117	115,862	73,293	70,242	80,115	80
Plantas	Abengoa Bioenergy Corp.	1,650,000	130,000	150,000	180,000	150,000	150,000	90,000	70,000	70,000	70,000	60
Associations	Southern Research Institute	1,610,000	50,000	40,000	50,000	50,000	50,000	40,000	50,000	50,000	50,000	30
Associations	Environmental Working Group	1,577,956	153,473	92,220	63,060	56,915	71,908	51,518	12,310	84,816	93,301	30

Figure 5-14: Bar chart and heat map visualizations. The heat map is created by color-coding the result of a sum formula in a calculated field (AMT) in a crosstab layout.

either their full precision or at least four significant digits, up to a maximum limit of six decimal places. When the layout system is used as part of our interactive visual query system, the full precision of numbers can still be seen in the formula bar, except when the number originates from a formula. This behavior mirrors that of Excel.

It is useful to display large numbers using a thousands separator character; for instance, in the US, the number 5129744.23 might be displayed as "5,129,744.23". We always do this by default, except for fields that, during the MEASURE phase, are only observed to contain integers of four digits or less. This avoids rendering years such as 2016 as "2,016". A similar policy exists in the SI Standard[4].

Bar charts and heat maps can be easily supported by allowing the magnitude of a numerical value to be indicated with a bar or a color, respectively. These formatting options can also be used in a crosstab to produce two-dimensional visualizations. See Figure 5-14. The Tableau/Polaris system uses crosstabs as a basis for a large number of similar two-dimensional visualizations [99].

5.5 Conclusion

We have presented a layout management algorithm that automates the display of structured nested data using the traditional visual idioms of hand-designed database UIs: tables, multi-column forms, and outline-style indented lists. By default, the widths of generated layouts are constrained so that only vertical scrolling may be necessary to view the data in its entirety. Our stylesheet system is further designed such that two input values mapping to the same schema field will always be styled in a similar way. Our hybrid layouts are 3.9 and 1.6 times more compact on average than outline layouts and horizontally unconstrained table layouts, respectively, and are as readable as table layouts even for large datasets. We believe that our system can function as a single output system for most of the data views commonly required in domain-specific database applications, whether they be large tables required to display information about many entities at once or form views that must display many details of a single entity compactly without scrolling.

[4] http://www.bipm.org/en/publications/si-brochure/section5-3-4.html

Chapter 6

Conclusion

We now discuss the extent to which the problems outlined in the Introduction has been solved, discuss future work, and summarize our contribution.

6.1 Discussion

6.1.1 Direct Manipulation

We have claimed that our visual query system satisfies the requirements of a *direct manipulation* interface. We here discuss the extent to which our own system satisfies the requirements of such a system, as described by Shneiderman [92] and, in the context of visual query systems, Liu and Jagadish [71] and Nandi et al. [78].

- *Divide query specification into progressive refinement step, and use intermediate results to help users formulate the query* [71]. This corresponds to our own requirement R1 (see Chapter 1), or a similar requirement as stated by Nandi et al.: *Users [may] manage, query, and manipulate data by directly interacting with it* [78]. This requirement is clearly satisfied by our own system, as illustrated by the example query building session from Section 1.1. SIEUFERD queries are built using a series of actions, each initiated by the user by means of interacting with data from the previous version of the query.

- *Continuously present data to users, after each data manipulation* [71]. This is a variation of the previous requirement. Again, SIEUFERD satisfies the requirement, always showing on the screen the result of each progressively refined query. Even when a query takes a long time to execute, SIEUFERD can immediately update the result layout to reflect a modified query, using stale data to fill the screen while waiting for updated results.

- *Rapid, reversible, incremental actions* [92]. As previously alluded to, all interactions in the SIEUFERD system happen at interactive speed, even if the underlying queries take a long time to execute. The speed of database queries depends on the size of the database, the indices available for query optimization, and the specific queries

constructed. In the data sources we used for our user studies, it was rare to see a query take more than a second to execute.

All query actions in the SIEUFERD system are reversible by means of undo/redo actions. Furthermore, undoing an action automatically cancels any long-running query that might be executing.

The concept of incremental actions was covered by the previous requirements.

- *Enable the user to modify an operation specified many steps earlier without redoing the steps afterwards* [71]. This is a more general variation of the reversibility requirement, corresponding to requirement R2 from our introduction (Chapter 1). It is not enough to allow the user to undo their way back to the operation that should be modified; the user must instead somehow be able to point to a representation of the operation and modify it directly. Arguably, this should be done without having to leave the direct manipulation interface. For instance, manually modifying a part of a generated SQL query or data manipulation script would not count as a solution here.

  In SIEUFERD, the solution to the problem of directly being able to modify queries is to associate every part of the query state with a column in the query's nested relational result. Each column then becomes an affordance for manipulating its own portion of the query state, such as a filter or a formula. Because we allow results to take the form of a nested relation rather than only a simple flat table, we are able to encode a very expressive query language into the schema of the result.

  While our current query manipulation actions work well for manipulating individual query operators such as filters and formulas, they are of limited use for more structural changes to a query, such as inserting a new relation between two existing relations in order to group on a subset of fields. This may be required for instance when doing aggregation with a custom grouping, or when preparing a query for use with a crosstab layout. An example was seen in Figure 5-11, where the TERMS relation was inserted between INSTRUCTORS and INSTRUCTORS_SECTIONS relations. Such queries are expressible using the existing query actions, but are awkward to construct[1], and may require operations to be redone if the user wishes to change the structure of the query, violating requirement R2. In the future, we propose to solve this problem by providing reversible higher-level query manipulation actions, such as GROUP and UNGROUP, that will rewrite the query to insert or extract relations according to common patterns. Note that these actions would act only as a symbolic manipulation of the current visual query; they do not require extensions to the SIEUFERD query model.

- *Visibility of the object of interest* [92]. Evaluating this requirement requires defining what exactly the *object of interest* is in context of a visual query language. Is it the *data* in the database, or the *query* being built by the user? If we consider the object of interest to be the query rather than the data, then many of the Diagram-based

[1]For example, in Figure 5-11, the TERMS relation is actually another instance of the INSTRUCTORS table, self-joined against its parent relation INSTRUCTORS on the table's primary key, with the formula TERM using an inward reference to a hidden child field of the COURSES relation containing the term of the course.

systems mentioned in Chapter 2 would qualify as direct manipulation systems, since the user is able to directly manipulate a diagram representing a query. We reject this definition, however, as the previous requirements make it clear that the user should be manipulating data in the database, for instance through the *result* of a query, rather than an abstract representation of a query in isolation. Looking at example data, and running intermediate versions of a query, is an essential part of the workflow for developing complex database queries. In systems that do not successfully integrate the manipulation of queries with the display of results, users will find themselves looking back and forth between the query representation on one side of the screen and a separate result representation on the other.

If the object of interest is the result of a query, then our system satisfies this requirement; the result of the current query is always visible on the screen at all times. However, our system goes one step further, also displaying the query that produced that result. The need to do so is not completely obvious; the systems in Chapter 2's Hidden Algebraic category, for instance, display the result of a query without representing the query itself somehow in the interface. As it turns out, displaying the query on the screen is a prerequisite for letting the user manipulate arbitrary parts of it, per the previous requirement.

With respect to the state of the current query, a possible point of objection is that our system does not show the *entire* state of the query on the screen at one time. Notably, showing the state of a filter, the definition of a formula, or the content of hidden columns requires the user to take additional action, such as opening the filter popup, moving the cell cursor, or opening the field selector, respectively. This is a conscious design decision, based on a tradeoff between completeness and conciseness of the visual state representation. In the case of filters and formulas, the use of partially hidden state is already well-established in spreadsheet interfaces. Hidden columns, while supported in spreadsheets, have a more prominent role in SIEUFERD due to the large number columns available for display in a typical database query. As discussed in connection with our user study, our system should include a more visible affordance for unhiding columns, without cluttering up every column header with extra indications.

We considered alternative designs for the field selection interface, including overlaying a ghost image of hidden columns onto the result layout during selection. However, this would make the list of field names hard to read, as they would be continue one after another on a single line, and also likely require horizontal scrolling. The current approach, with the names of available fields being listed vertically, was chosen for its compactness and readability. It is also easy to quickly unhide a number of fields to see the data in each, exactly where it would appear in the result layout. Any undesired fields can then be hidden again, without reentering the field selector, by means of the HIDE action. Note that the root field displayed in the field selector is based on the field for which the FIELDS action was invoked, allowing the user to go directly to the desired relation to rather than having to navigate the field hierarchy from within the field selector.

- *The display should indicate a complete image of what the current status is, what errors have occurred, and what actions are appropriate* [92]. The completeness of the state representation was discussed in connection with the previous requirement. With regard to error states, the SIEUFERD interface includes a facility for communicating error messages to the user from within the direct manipulation interface, as was shown in Figure 3-3. The context menu acts as a complete list of actions that can be invoked during query construction, with column header icons indicating the presence of existing state that can be modified. One thing missing is a better affordance for opening the context menu in the first place. In the future, we might show a spreadsheet-style filter dropdown button whenever the user hovers over a column.

- *Replacement of complex command language syntax by direct manipulation of the object of interest* [92]. Our system replaces manipulation of textual SQL queries with direct manipulation of data in the database.

  One possible objection is our system's use of formula expressions, a textual language. The scope of the language, however, is limited to arithmetic expressions, and except for the use of column-based references, the language is similar to that of spreadsheet formulas. Like in a spreadsheet, the editing of formulas is done by direct manipulation of the location where the formula's result will end up, and references can be inserted by clicking or keyboard-selecting the column to reference. The use of column-based references should also make formulas simpler to read than spreadsheet formulas. We do not believe that the use of formulas disqualify spreadsheets from being considered a direct manipulation interface. In fact, the original VisiCalc spreadsheet, for which formula calculations was the defining feature, is one of Shneiderman's original examples of a direct manipulation interface [92]. The author believes the textual expression Price * (1 - Discount) is in fact the best possible way to visualize the calculation Price * (1 - Discount). That said, it would be useful to have an explicit Sum action to help users create the formula for this particularly common use case.

  We considered giving aggregate functions a special representation in the SIEUFERD query model, for instance as an optional decoration that could be applied to any numerical column. The hope would be that the formula language could then omit aggregate functions entirely. However, the formula language would then have required a special syntax for referencing the aggregate result of a column, negating any simplifications. Besides, aggregate functions in formulas are a well-established spreadsheet concept. We thus decided on the current design.

6.1.2 Expressiveness of the Visual Query Language

We have already evaluated the formal expressiveness of our visual query language. The expressive power is equivalent to that SQL-92, plus the ability to generate nested results. By achieving SQL-like expressiveness from a true direct manipulation interface, we believe SIEUFERD has solved the visual query language problem.

6.1.3 Use and Readability of Layouts

During user studies involving visual query building, we have only so far used the layout manager to produce plain nested table layouts. We believe some more work is required before the form-style hybrid layouts can be used effectively during query construction.

The first improvement needed is to actually provide a user interface for switching between form and table layouts. Rather than showing a single query result using two different layouts, we believe the form layout could be used as a detail pane for the table layout, showing more information for a single selected row. This requires some changes in how state is stored in the user interface, as the two panes will now be showing a different selection of fields. It may also be desirable to jump from one detailed form layout to another, to drill down on specific entities, or to magnify a table in a form layout to a full table layout. This requires both design and implementation work.

A second improvement relates to the readability of form-style hybrid layouts. For better readability, the system should track which field or fields in each relation best represents a human-readable heading for items in that relation, and display these headers in larger font. The font of the header should decrease with the depth of each relation. Improvements should also be made to the appearance of outline columns in hybrid layouts, as they currently make layouts hard to read.

Relating to both table and form layouts, complicated queries may contain many levels of relations, leading to a large number of field labels being stacked on top of each other. It could be useful for the user to have a way to hide or collapse labels to make the query result more readable. The downside is that selecting a relation's label is currently the only way to initiate certain actions, such as filtering on the relation in question. More generally, it would be useful for the user to be able to customize layout formatting details. This is currently possible only via a debugging interface.

6.2 Future Work

6.2.1 Query Interface

In the current query interface, some queries are expressible yet awkward to construct; examples include greater-than/less-than conditions, grouping on custom attributes, and `UNION`-type queries. Here, the interface could be improved without significant changes to the underlying query model. For instance, we could include support for range filters, explicit *Group*/Ungroup rewriting actions (discussed in the previous section), and the previously proposed syntactic sugar for unions (Figure 4-6), respectively.

We would also like to implement Section 3.3.2's recommendations related to working with databases with a large number of fields per table.

We would like to implement a type system for columns and formulas. This will improve error messages for user formulas. It is also a prerequisite for editing support (see below).

Despite the theoretical idea that all query optimization should be left to the database backend, there are many ways to generate the SQL queries required for evaluating a SIEUFERD query, some of which optimize better than others. One area of future work

involves the implementation and evaluation of various optimizations in the generated SQL queries.

6.2.2 The CRUD Application Use Case

In our introduction, we mentioned the various kinds of views that are found in typical Create-Read-Update-Delete (CRUD) applications. Our visual query interface, in combination with our automatic layout generator, can produce all of these views. Table views are covered by our layout manager's nested table layouts, forms and reports are covered by our layout manager's hybrid layouts, while search forms can be replaced by filters opened in hybrid layouts. But as mentioned in the previous section, our system currently lacks a user interface for managing and navigating between these views. This problem needs to be solved before our system can hope to replace CRUD applications.

More importantly for the CRUD application use case, all views generated by our visual query system are currently read-only. In the future, we hope to incorporate *editing* of data. The semantics of our visual query language are already well-suited for producing updatable views, and the automatic form layouts produced by our layout algorithm can serve as a good user interface for the common task of editing individual entities in the database and all their related information. The ability to edit data would allow SIEUFERD to act as a complete schema-independent end user front-end for relational databases.

6.3 Conclusion

This thesis has presented SIEUFERD, a visual query system. The system's visual query language is the first to support both the specification and subsequent modification of arbitrary SQL queries from within a pure direct manipulation interface. The system's graphical output engine is the first to fully automate the generation of nested table-, form- and report-style layouts based on observed statistical measurements of the data in query results. The complete system allows end-users to produce all output displays commonly found in tailored CRUD database applications, using a small set of spreadsheet-like operations.

By directly manipulating nested relational results, the user can express a relationally complete set of query operators plus calculation, aggregation, outer joins, sorting, and nesting. This covers the full set of query operators generally considered as the minimum to model SQL, and expresses, for example, all `SELECT` statements valid in SQL-92. At the same time, the user always remains able able to track and modify the state of the complete query. 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 that SIEUFERD's interaction style holds promise as an alternative to hand-coded SQL.

Bibliography

[1] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Ré, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. The Beckman report on database research. *Communications of the ACM*, 59(2):92–99, January 2016.

[2] Azza Abouzied, Joseph Hellerstein, and Avi 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.

[3] Stefan 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.

[4] Christopher Ahlberg. *Dynamic Queries*. PhD thesis, Institutionen för datavetenskap, Chalmers tekniska högskola, Göteborg, Sweden, 1996.

[5] Christopher Ahlberg and Ben Shneiderman. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '94)*, pages 313–317, New York, NY, USA, 1994. ACM.

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

[7] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System R: A relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[8] Eirik Bakke and Edward 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.

[9] Eirik Bakke and David R. Karger. Expressive query construction through direct manipulation of nested relational results. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*, pages 1377–1392, New York, NY, USA, 2016. ACM.

[10] Eirik Bakke, David R. Karger, and Robert C. Miller. A spreadsheet-based user interface for managing plural relationships in structured data. In *Proceedings of the 29th International Conference on Human Factors in Computing Systems (CHI '11)*, pages 2541–2550, New York, NY, USA, 2011. ACM.

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

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

[13] Benjamin B. Bederson, Ben Shneiderman, and Martin Wattenberg. Ordered and quantum treemaps: Making effective use of 2D space to display hierarchies. *ACM Transactions on Graphics (TOG)*, 21(4):833–854, October 2002.

[14] Tim Berners-Lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David 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.

[15] Chris Bizer and Andreas Schultz. The Berlin SPARQL benchmark. *The International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2), 2009.

[16] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, and Jean Vanderdonckt. Towards a dynamic strategy for computer-aided visual placement. In *Proceedings of the workshop on Advanced Visual Interfaces (AVI '94)*, pages 78–87, New York, NY, USA, 1994. ACM.

[17] John Brooke. SUS: A quick and dirty usability scale. In Patrick W. Jordan, Bruce Thomas, Bernard A. Weerdmeester, and Ian L. McClelland, editors, *Usability evaluation in industry*, pages 189–194. Tailor & Francis, London, UK, 1996.

[18] Josep Maria Brunetti, Roberto García, and Sören Auer. From overview to facets and pivoting for interactive exploration of semantic web data. *International Journal of Semantic Web & Information Systems*, 9(1):1–20, January 2013.

[19] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry 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.

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

[21] Stuart K. Card, Thomas P. Moran, and Allen Newell. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):396–410, July 1980.

[22] Tiziana Catarci, Maria F. Costabile, Stefano Levialdi, and Carlo Batini. Visual query systems for databases: A survey. *Journal of Visual Languages & Computing*, 8(2):215–260, 1997.

[23] Jonathan P Caulkins, Erica Layne Morrison, and Timothy Weidemann. Spreadsheet errors and decision making: Evidence from field interviews. *Journal of Organizational and End User Computing*, 19(3):1, 2007.

[24] Yolande E. Chan and Veda C. Storey. The use of spreadsheets in organizations: Determinants and consequences. *Information & Management*, 31(3):119–134, 1996.

[25] Kerry Shih-Ping Chang and Brad 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.

[26] Peter Pin-Shan Chen. The Entity-Relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[27] Woei-Kae Chen and Pin-Ying Tu. VisualTPL: A visual dataflow language for report data transformation. *Journal of Visual Languages & Computing*, 25(3):210–226, 2014.

[28] Ed Huai-Hsin Chi, P. Barry, J. Riedl, and J. Konstan. A spreadsheet approach to information visualization. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis '97)*, pages 17–24, 1997.

[29] Richard Chimera and Ben Shneiderman. An exploratory evaluation of three interfaces for browsing large hierarchical tables of contents. *ACM Transactions on Information Systems (TOIS)*, 12:383–406, October 1994.

[30] Petr Chmelar, Radim Hernych, and Daniel Kubicek. Interactive visualization of data-oriented XML documents. In Tarek Sobh, editor, *Advances in Computer and Information Sciences and Engineering*, pages 390–393. Springer Netherlands, 2008.

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

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

[33] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[34] Gautam Das, Vagelis Hristidis, Nishant 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.

[35] Chris J. Date, Burt Grad, and Thomas Haigh. Oral history of C. J. Date. In *Oral History Collection*. Computer History Museum, Mountain View, CA, USA, June 2007.

[36] Emilia 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.

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

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

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

[40] Mary Fernández, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. Silkroute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems (TODS)*, 27(4):438–493, 2002.

[41] Krzysztof Gajos and Daniel S. Weld. SUPPLE: Automatically generating user interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI '04)*, pages 93–100, New York, NY, USA, 2004. ACM.

[42] Roberto García, Rosa Gil, Juan Manuel Gimeno, Eirik Bakke, and David R. Karger. Besdui: A benchmark for end-user structured data user interfaces. In *Proceedings of the 15th International Semantic Web Conference (ISWC '16), to appear*. Springer, 2016.

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

[44] M. Ghoniem, J. Fekete, and P. Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis '04)*, pages 17–24, 2004.

[45] Martin Graham and Jessie Kennedy. A survey of multiple tree visualisation. *Information Visualization*, 9(4):235–252, 2010.

[46] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–53, 1997.

[47] Yanbo Han, Guiling Wang, Guang Ji, and Peng Zhang. Situational data integration with data services and nested table. *Service Oriented Computing and Applications*, 7(2):129–150, 2013.

[48] G. D. Held, M. R. Stonebraker, and E. Wong. INGRES: A relational data base system. In *Proceedings of the National Computer Conference and Exposition (AFIPS '75)*, pages 409–416, New York, NY, USA, 1975. ACM.

[49] Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. *Journal of the ACM (JACM)*, 48(4):880–907, July 2001.

[50] Geert-Jan Houben and Jan 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.

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

[52] Charles Jacobs, Wilmot Li, Evan Schrier, David Bargeron, and David Salesin. Adaptive grid-based document layout. *ACM Transactions on Graphics (TOG)*, 22(3):838–847, July 2003.

[53] 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.

[54] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong 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.

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

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

[57] Magesh 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.

[58] Josef Jelinek and Pavel Slavik. XML visualization using tree rewriting. In *Proceedings of the 20th Spring Conference on Computer Graphics (SCCG '04)*, pages 65–72, New York, NY, USA, 2004. ACM.

[59] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd Conference on Visualization (VIS '91)*, pages 284–291, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[60] Minsuk Kahng, Shamkant B. Navathe, John T. Stasko, and Duen Horng Chau. Interactive browsing and navigation in relational databases. *Computing Research Repository/arXiv*, abs/1603.02371, 2016.

[61] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2917–2926, December 2012.

[62] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey 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.

[63] Eser Kandogan, Eben Haber, Rob Barrett, Allen Cypher, Paul Maglio, and Haixia 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.

[64] William Kent. A simple guide to five normal forms in relational database theory. *Communications of the ACM*, 26:120–125, February 1983.

[65] Henry F. Korth and Mark A. Roth. Query languages for nested relational databases. In S. Abiteboul, P. Fischer, and H. Schek, editors, *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*, pages 190–204. Springer Berlin/Heidelberg, 1989.

[66] Keith Kowalzcykowski, Alin Deutsch, Kian Win Ong, Yannis Papakonstantinou, Kevin Keliang Zhao, and Michalis Petropoulos. Do-It-Yourself database-driven web applications. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR '09)*, 2009.

[67] Dariusz Król, Jacek Oleksy, Małgorzata Podyma, and Bogdan 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.

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

[69] James R. Lewis and Jeff 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.

[70] Leonid Libkin and Limsoon Wong. On the power of aggregation in relational query languages. In Sophie Cluet and Rick 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.

[71] Bin 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.

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

[73] Jock D. Mackinlay, Pat Hanrahan, and Chris Stolte. Show Me: Automatic presentation for visual analysis. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 13(6):1137–1144, November/December 2007.

[74] Richard Matthew McCutchen, Shachar Itzhaky, and Daniel Jackson. Initial report on Object Spreadsheets. Technical Report MIT-CSAIL-TR-2016-001, MIT Computer Science and Artificial Intelligence Laboratory, January 2016.

[75] Nancy McDonald and Michael Stonebraker. CUPID–the friendly query language. In *Proceedings of the ACM Pacific 75 Conference*, pages 127–131, 1975.

[76] 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.

[77] Richard Mitchell, David Day, and Lynette Hirschman. Case study: Fishing for information on the Internet. In *Proceedings of the First Information Visualization Symposium (InfoVis '95)*, pages 105–111, Los Alamitos, CA, USA, October 1995. IEEE Computer Press.

[78] Arnab Nandi, Lilong Jiang, and Michael Mandel. Gestural query specification. *Proceedings of the VLDB Endowment*, 7(4):289–300, 2013.

[79] Quang Vinh Nguyen and Mao Lin Huang. EncCon: An approach to constructing interactive visualization of large hierarchical data. *Information Visualization*, 4(1):1–21, 2005.

[80] Raymond R. Panko and Salvatore Aurigemma. Revising the Panko–Halverson taxonomy of spreadsheet errors. *Decision Support Systems*, 49(2):235–244, 2010.

[81] Yannis Papakonstantinou, Michalis Petropoulos, and Vasilis 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.

[82] Jonathan D. Pemberton and Andrew J. Robson. Spreadsheets in business. *Industrial Management & Data Systems (IMDS)*, 200(8):379–388, 2000.

[83] Robert Pienta, Acar Tamersoy, Alex Endert, Shamkant Navathe, Hanghang Tong, and Duen Horng Chau. VISAGE: Interactive visual graph querying. In *Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI '16)*, pages 272–279, New York, NY, USA, 2016. ACM.

[84] Stavros Polyviou, George Samaras, and Paraskevas 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.

[85] Stephen G. Powell, Kenneth R. Baker, and Barry Lawson. A critical review of the literature on spreadsheet errors. *Decision Support Systems*, 46(1):128–138, December 2008.

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

[87] Neil Raden. Shedding light on shadow IT: Is Excel running your business? Technical report, Hired Brains, Inc., January 2005.

[88] Ramana Rao and Stuart K. Card. The table lens: Merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '94)*, pages 318–322, New York, NY, USA, 1994. ACM.

[89] Thomas Reschenhofer and Florian Matthes. An empirical study on spreadsheet shortcomings from an information systems perspective. In *Proceedings of the 18th International Conference on Business Information Systems (BIS '15)*, pages 50–61, Cham, Switzerland, June 2015. Springer International Publishing.

[90] George Robertson, Kim Cameron, Mary Czerwinski, and Daniel Robbins. Polyarchy visualization: Visualizing multiple intersecting hierarchies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '02)*, pages 423–430, New York, NY, USA, 2002. ACM.

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

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

[93] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[94] Ahmet Soylu, Martin Giese, Ernesto Jimenez-Ruiz, Guillermo Vega-Gorgojo, and Ian Horrocks. Experiencing OptiqueVQS: A multi-paradigm and ontology-based visual query system for end users. *Universal Access in the Information Society*, 15(1):129–152, March 2016.

[95] 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.

[96] Michael Spenke, Christian Beilken, and Thomas Berlage. Focus: The interactive table for product comparison and selection. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology (UIST '96)*, pages 41–50, New York, NY, USA, 1996. ACM.

[97] Hennie J. Steenhagen, Peter M. G. Apers, and Henk 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.

[98] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 8(1):52–65, 2002.

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

[100] Christopher Richard Stolte. *Query, analysis, and visualization of multidimensional databases*. PhD thesis, Stanford University, Stanford, CA, USA, 2003.

[101] Michael Stonebraker and Joseph M. Hellerstein. What goes around comes around. In Joseph M. Hellerstein and Michael Stonebraker, editors, *Readings in Database Systems*. The MIT Press, Cambridge, MA, USA, 4th edition, 2005.

[102] Keishi Tajima and Kaori Ohnishi. Browsing large HTML tables on small screens. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology (UIST '08)*, pages 259–268, New York, NY, USA, 2008. ACM.

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

[104] Jerzy 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.

[105] Jan Van den Bussche and Stijn 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.

[106] Amy Voida, Ellie Harmon, and Ban Al-Ani. Homebrew databases: Complexities of everyday information management in nonprofit organizations. In *Proceedings of the Annual Conference on Human Factors in Computing Systems (CHI '11)*, pages 915–924, New York, NY, USA, 2011. ACM.

[107] Lutz Wegner, Sven Thelemann, Jens Thamm, Dagmar Wilke, and Stephan Wilke. Navigational exploration and declarative queries in a prototype for visual information systems. In Clement Leung, editor, *Visual Information Systems*, volume 1306 of *Lecture Notes in Computer Science*, pages 199–218. Springer Berlin/Heidelberg, 1997.

[108] Fan Yang, Nitin Gupta, Chavdar Botev, Elizabeth F Churchill, George Levchenko, and Jayavel Shanmugasundaram. WYSIWYG development of data driven web applications. *Proceedings of the VLDB Endowment*, 1(1):163–175, 2008.

[109] Shengdong Zhao, Michael J. McGuffin, and Mark H. Chignell. Elastic hierarchies: Combining treemaps and node-link diagrams. In *IEEE Symposium on Information Visualization (InfoVis '05)*, pages 57–64, October 2005.

[110] Caroline Ziemkiewicz, R. Jordan Crouser, Ashley Rye Yauilla, Sara L. Su, William Ribarsky, and Remco Chang. How locus of control influences compatibility with visualization style. In *2011 IEEE Conference on Visual Analytics Science and Technology (VAST '11)*, pages 81–90, October 2011.

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