Organic Databases

H.V. Jagadish, Arnab Nandi, and Li Qian*

University of Michigan
Ann Arbor, MI, USA

Abstract. Databases today are carefully engineered: there is an expensive and deliberate design process, after which a database schema is defined; during this design process, various possible instance examples and use cases are hypothesized and carefully analyzed; finally, the schema is ready and then can be populated with data. All of this effort is a major barrier to database adoption.

In this paper, we explore the possibility of *organic* database creation instead of the traditional engineered approach. The idea is to let the user start storing data in a database with a schema that is just enough to cove the instances at hand. We then support efficient schema evolution as new data instances arrive. By designing the database to evolve, we can sidestep the expensive front-end cost of carefully engineering the design of the database.

The same set of issues also apply to database querying. Today, databases expect queries to be carefully specified, and to be valid with respect to the database schema. In contrast, the organic query specification model would allow users to construct queries incrementally, with little knowledge of the database. We also examine this problem in this paper.

1 Motivation

Database technology has made great strides in the past decades. Today, we are able to process efficiently ever larger numbers of ever more complex queries on ever more humongous data sets. We can be justifiably proud of what we have accomplished.

However, when we see how information is created, accessed, and shared today, database technology remains only a bit player: much of the data in the world today remains outside database systems. Even worse, in the places where database systems are used extensively, we find an army of database administrators, consultants, and other technical experts all busily helping users get data into and out of a database. For almost all organizations, the indirect cost of maintaining a technical support team far exceeds the direct cost of hardware infrastructure and database product licenses. Not only are support staff expensive, they also interpose themselves between the users and the databases. Users cannot interact with the database directly and are therefore less likely to try less straightforward operations. This hidden opportunity cost may be greater than the visible costs of hardware/software and technical staff. Most of us remember the day not too long ago when booking a flight meant calling a travel agent who used magic incantations at an arcane system to pull up information regarding flights and to make bookings. Today, most of us book our own flights on the web through interfaces that

* Supported in part by NSF under grant IIS-1017296.

S. Kikuchi et al. (Eds.): DNIS 2011, LNCS 7108, pp. 49–63, 2011.

are simple enough for anyone to use. Many enjoy the power of being able to explore options for themselves that would have been too much trouble to explain to an agent, such as willingness to trade off price against convenience of a flight connection.

Search engines have done a remarkable job at directly connecting users with the web. Users can publish documents of any form on the Web. For a keyword query, the user is pointed to a set of documents that are most likely to be relevant to the user. This best-effort nature can lead to possibly inaccurate results, but it allows the users the ability to easily and efficiently get information into and out of the ever-changing Web.

In contrast, the database world has had the heritage of constructing rigid, *precisely* defined, carefully *planned*, explicitly engineered, silos of information based on *predictions* regarding data and queries. It was assumed that information would be clean, rigid and well structured. This has led to databases today being hard to design, hard to modify, and hard to query.

When we look at characteristics of search, we find that there is very low prediction and planning burden placed on users – neither to query nor to publish data. Furthermore, precision, while desirable, is not required. In contrast, users interacting with databases find themselves fighting an uphill battle with the constant flux of the data they deal with in today's highly connected world.

Our goal in this paper is to render database interaction lenient in its demands for prediction, planning, and precision. We call this *organic*, to distinguish from the carefully designed and engineered "synthetic" database and query system used today. The result of an organic query may not be as perfect as the result of an engineered query, but it has the benefit of not requiring precision and planning, and hence being more "natural" for most users. To be able to develop such an organic system, let us first study the precision and planning challenges that users face as they interact with databases.

2 Challenges

2.1 Structure Specification Challenge

Precise specification is challenging for users interacting with a database. Consider an airline database with a basic schema shown in Figure 1, for tracing planes and flights. The data encapsulated is starting location, destination, plane information, and times — essentially what every passenger thinks of as *a flight*. Yet, in our normalized relational representation, this single concept is recorded across four different tables. Such splattering of data decreases the usability of the database in terms of schema comprehension, join computation, and query expression.

First, given the large number of tables in a database, often with poorly named entities, it is usually not easy to understand how to locate a particular piece of data. Even in a toy schema such as Figure 1, there is the possibility of trouble. Obviously, the *airports* table has information about the starting location and the destination. To find what is used by a particular flight, we have to bring up the schema and follow the foreign key constraint, or trace the database creation statements. Neither solution is user-friendly, and thus the current solution is often to leave the task to DBAs.

The next problem users face is computing the joins. We break apart information during the database design phase such that everything is normalized — space efficient

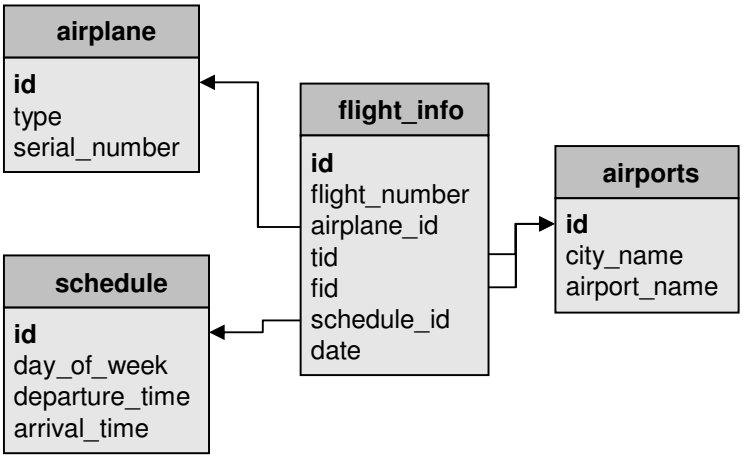

Fig. 1. The base tables needed to store a "flight". A flight contains from location, destination, airplane info and schedule, yet consists of at least four tables. Note that an actual schema for such data is likely to involve many more attributes and tables.

and amenable to updates. However, the users will have to stitch the information back together to answer most real queries. The fundamental issue is that joins destroy the connections between information pertaining to the same real world entities. Query specification is non-intuitive to most normal users in consequence. But even the design is brittle. What if a single flight has multiple flight numbers on account of code sharing? What about special flights not on a weekly schedule? There are any number of such unanticipated possibilities that could render a carefully designed structure inadequate instantly.

2.2 Remote Specification Challenge

Querying in its current form requires *prediction* on the part of the user. In our airline database example, consider the specification of a three letter airport code. Some interfaces provide a drop down list of all the cities that the airline flies into. For an airline of any size, this list can have hundreds of entries, most of which are not relevant to the user. The fact that it is alphabetized may not help — there may be multiple airports for some major cities, the airport may be named for a neighboring city, and so on.

A better interface allows a user to enter the name of the place they want to get to, and then looks for close matches. This cannot be a simple string comparison — we need Narita airport to be suggested no matter whether the user entered Narita or Tokyo or even Tokyu. This does not seem too hard, and some airline web sites will do this. But now consider a user who wants to visit Aizu. No airline search interface today, to our knowledge, can suggest flying into Narita airport in response to a search for Aizu airport even though that is likely to be the preferred solution for most travelers.

On account of difficulty in prediction, it is often the case that the user does not initially specify the query correctly. The user then has to revise her query and resubmit

if it did not return desired results. However, essentially all query languages, including visual query builders, separate query specification from output.

Our goal is to enable users to query a database in a WYSIWYG (What You See Is What You Get) fashion. Consider the display of a world map. The user could zoom into the area of interest and select airports geographically from the choices presented. Most map databases today provide excellent direct manipulation capabilities, including pan, zoom, and so on. Imagine a map database without these facilities that requires users to specify, through a text selection of zip code or latitude/longitude, the portion of the map that is of interest each time. We would find it terribly frustrating. Unfortunately, most database query interfaces today are not WYSIWYG and can be compared to this hypothetical frustrating map query interface.

What does WYSIWYG mean for databases? After all, the point of specifying a query is to get information that the user does not possess. Even search engines are not WYSIWYG. A WYSIWYG interface for selection specification and data results involves a constant *predictive* capability on the part of the system. For example, instantaneous-response interfaces (56) allow users to gain insights into the schema and the data during query time, which allows the user to continuously refine the query *as they are typing the initial query*. By the time the user has typed out the entire query, the query has been correctly formulated and the results have returned. Furthermore, if the user then wishes to modify the query, this should be possible by direct manipulation of the result set rather than an *ab initio* restatement of the query.

2.3 Schema Evolution Challenge

While database systems have fully established themselves in the corporate market, they have not made a large impact on how users organize their everyday information. Many users would like to put into their databases (8) information such as shopping lists, expense reports, etc. The main reason for this is that creating a database is not easy.

Database systems require that the schema be specified in advance, and then populated with data. This burdens the user with developing an abstract design of the schema – without any concrete data – a task that we computer scientists are trained to do, but most others find very difficult. Furthermore, careful planning is required as users are expected to predict what data they will need to store in the future, and what queries they may ask, and use these predictions to develop a suitable schema.

Example 21. *Consider a user, Jane, who started to keep track of her shopping lists. The first list she created simply contained a list of items and quantities of each to be purchased. After the first shopping trip, Jane realized that she needed to add price information to the list to monitor her expenses and she also started marking items that were not in stock at the store. A week before Thanksgiving, Jane created another shopping list. However, this time, the items were gifts to her friends, and information about the friends therefore needed to be added to create this "gift list." A week after Christmas, Jane started to create another "gift list" to track gifts she received from her friends. However, the friends information were now about friends giving her gifts. In the end, what started as a simple list of items for Jane had become a repository of items, stores, and more importantly, friends — an important part of Jane's life.*

The above example, although simple, illustrates how an everyday database evolves and the many usability challenges facing a database system. First, users do not have a clear knowledge of what the final structure of the database will be and therefore a comprehensive design of the database is impossible at the beginning. For example, Jane did not know that she needed to keep track of information about her friends until the time had come to buy gifts for them. Second, the structure of the database grows as more information become available. For example, the information about price and out of stock only became available after the shopping trip. Finally, information structures may be heterogeneous. For example, the two "gift lists" that Jane created had different semantics in their friends information and the database needs to gracefully handle this heterogeneity.

In summary, for everyday data, the structure grows incrementally and a database system must provide interfaces for users to easily create both unstructured and structured information and to fluidly manipulate the structure when necessary.

3 Proposed Solution

3.1 Presentation Data Model

We propose the use of a *presentation* data model (36), as a full-fledged layer above the physical and logical layers in the database. Just as the logical layer provides data abstraction and saves the user from having to worry about physical data aspects such as data structures, indices, access methods, etc., the presentation layer saves the user from having to worry about logical data aspects such as relational structure, keys, joins, constraints, etc. To do this, the presentation layer should be able to represent data in a form most suited for the user to easily comprehend, manipulate and query.

3.2 Addressing Structure Specification Challenge

We address the structure specification challenge through the *qunit* search paradigm (57), where the database is translated into a collection of independent qunits, which can be treated as documents for standard IR-like document retrieval. A qunit is the basic, independent semantic unit of information in a database. It represents a quantified unit of information in response to a user's query. The database search problem then becomes one of choosing the most appropriate qunit(s) to return, in ranked order. Users only have to input keywords, which is much simpler than navigating complex database schema and specifying a structured query. In other words, the precision burden is lifted from the user. Consider the flight example in Figure 1. A qunit "flight" can be defined to represent the complete information of what a passenger thinks of as a flight. The qunit includes starting location, destination, plane, and time of travel. This completely relieves users from having to manually performing joins among all the tables. As a user inputs a search criterion, for example "from DTW to LAX, Jan. 2010", qunits are ranked based on the input and the best matches are presented to the user.

We now explain the definition of qunits over a database, and how to search based on qunits. We use a slightly more complex *IMDb* movie database in order to explain more effectively. Figure 2 (a) shows a simplified example schema of a movie database, which

(a) An Simplified database schema

(b) Qunit Search on IMDb

Fig. 2. Qunit Example

contains entities such as movie, cast, person, etc. Qunits are defined over this database corresponding to various information needs. For example, we can define a qunit "cast", as the people associated with a movie. Meanwhile, rather than having the name of the movie repeated with each tuple, we may prefer to have a nested presentation with the movie title on top and one tuple for each cast member. The base data in IMDb is relational, and against its schema, we would write the base expression in SQL with the conversion expression in XSL-like markup as follows:

```
SELECT * FROM person, cast, movie
WHERE cast.movie_id = movie.id AND
cast.person_id = person.id AND
movie.title = "$x"
RETURN
<cast movie="$x">
<foreach:tuple>
<person>$person.name</person>
</foreach:tuple>
</cast>
```

The combination of these two expressions forms our qunit definition. On applying this definition to a database, we derive qunit instances, one per movie.

To search based on qunits, consider the user query, *star wars cast*, as shown in Figure 2 (b). Queries are first processed to identify entities using standard query segmentation techniques (73). In our case one high-ranking segmentation is "[movie.name] [cast]" and this has a very high overlap with the qunit definition that involves a join between "movie.name" and "cast". Now, standard IR techniques can be used to evaluate this query against qunit instances of the identified type; each considered independently even if they contain elements in common. The qunit instance describing the cast of the movie Star Wars is chosen as the appropriate result.

In current models of keyword search in databases, several heuristics are applied to leverage the database structure to construct a result on the fly. These heuristics are often based on the assumption that the structure within the database reflects the semantics

assumed by the user (though data / link cardinality is not necessarily an evidence of importance), and that all structure is actually relevant towards ranking (though internal id fields are never really meant for search).

3.3 Addressing Schema Evolution Challenge

In this section we address the schema evolution challenge (Sec. 2.3) by proposing a technique for drag-and-drop modification of data schemas in the spreadsheet-like presentation model, enabling organic evolution of a schema and lifting the planning burden from the user. Consider the example of Jane's shopping list again. Figure 3 shows how Jane can organically grow the schema of the shopping list table. Initially, she has only columns for items to shop (Figure 3 (a)). She later tries to add information about friends to whom the gifts will be given, for instance, by adding a "name" column in "Shopping List". But now, Peter, a close friend of Jane, appears twice since both item Xbox and iPod will be given to him. As a result, Jane may think it makes more sense to group the gifts by person. Jane can do this by dragging the header of the name column and dropping it on the lower edge of the "Shopping List" (Figure 3 (b)). This makes the name attribute a level up; the rest of the columns forms a sub-relation "Gift" (shown in Figure 3 (c)). Now Jane can feel free to add new information, such as an attribute "address", for her friends without worrying that these information would be duplicated (Figure 3 (d)). This process shows how effortless it is for Jane to grow the table about shopping items to include information about friends and structure the table as she desires.

Shopping List

Item	Quantity	Price	In Stock
Xbox	1	300	N
Swarovski	1	200	Y
iPod	2	180	Y

(a) Initial Shopping List

Shopping List

Item	Quantity	Price	In Stock	Name
Xbox	1	300	N	Peter
Swarovski	1	200	Y	Cathy
iPod	2	180	Y	Peter

(b) Moving Name Column

Shopping List

Name	Gift			
	Item	Quantity	Price	In Stock
Peter	iPod	2	180	Y
	Xbox	1	300	N
Cathy	Swarovski	1	200	Y

(c) After Moving Name Column

Shopping List

Name	Address	Gift			
		Item	Quantity	Price	In Stock
Peter	2364 Plymouth	iPod	2	180	Y
		Xbox	1	300	N
Cathy	1056 Bishop	Swarovski	1	200	Y

(d) Adding Address Column

Fig. 3. Organic Schema Evolution

Next, we briefly outline the challenges in building a system such as this, and our plans to tackle these challenges.

Specification: Specifying a schema update as in Figure 3 is challenging using existing tools. For example, using conventional spreadsheet software, it is impossible to arrive at a hierarchical schema as shown in Figure 3 (d). To specify the schema update, one has to split the table manually. Alternatively, using a relational DBMS, one has to set up the cross-table relationship, which is not easy for end-users, even with support from GUI tools.

We show how to use a *presentation layer* to address the specification challenge. We design the presentation layer based on a next-generation spreadsheet and it supports

easy schema creation and modification through a simple drag-and-drop interface. We call such a spreadsheet *span table* because it is presented in such a way that both table headers and data fields can span multiple cells. The presentation supports four key operations: move an attribute to be part of a sub-relation (e.g., we can move the "Name" column back to "Gift" in Figure 3 (d)), move an attribute out of a sub-relation (the converse of the previous one), create a intermediate sub-relation by moving an attribute *up* one layer (e.g., Jane moves "Name" out to create a new sub-relation under "Shopping List" as in Figure 3 (b)) or *down* a layer (e.g., moving "In Stock" down deepens it by inserting a new immediate level, with only "In Stock" in it; Jane can later add new columns such as "Date" to indicate the timestamp of stocking information).

Data Migration: Once a new schema is specified, there is still a critical task of migrating existing data to the new schema. Because the schema structure is changed, one has to introduce a complex mapping in order to "fit" the old data into the new schema. Even if spreadsheet software supporting hierarchical schema is provided, the user may still have to manually copy data in a cell-by-cell manner to perform such mapping, which is extremely time-consuming and error-prone.

We address this challenge with an *algebraic layer*. Directly below the presentation layer, the algebraic layer must translate drag-and-drops into operations that modify the basic structure of the span table. For this purpose, we have proposed a novel *span table algebra* consisted of three sets of operations. The first set, schema restructuring operators, corresponds to the four aforementioned operations in the presentation layer. We also have a second set of schema modification operators for adding/dropping columns in any sub-relations. Finally, there is a set of data manipulation operators (insert, delete, and update), which extends traditional data edit to our hierarchical presentation. This algebraic layer completely automates the data migration as soon as the the schema modification is performed.

Data Integrity: Expressing and understanding integrity constraints is central to schema design, and thus also critical for an organic database where schema is continuously evolved. Functional dependencies (FD) are often used in database design to add semantics to schemas and to assert integrity constraints for data.

Nested functional dependencies have been studied extensively in the past (32). However, CRIUS presents some new challenges due to its user-centric support for data and schema modification. When a user updates data, or modifies the schema, it is important to understand how the update affects existing dependencies so that we can communicate this information back to the user, and optionally take steps to resolve any resulting inconsistencies.

For this challenge, we consider two specific operations: data value updates and schema updates. For the former case, we show how data value updates and integrity constraints interfere with each other and how we may take advantage of such inference to guide user data entry from a set of appropriately maintained FDs. Specifically, we feature autocompletion for qualified data entry and provide a contextual menu to alert the user each time she issues an update that violates a given FD, in order to preserve data integrity. For schema updates, our hypothesis is that for each schema update

operation there is a way to "rewrite" involved FDs to preserve their validity. Precisely how to rewrite the schema is described in detail in (62).

Performance: Schema evolution is usually a heavy-weight operation in traditional database systems. It is not unusual for a commercial database to take days to complete the maintenance required after schema evolution. IT organizations carefully plan schema changes, and make them only infrequently. In contrast, everyday users are unlikely to plan carefully. We would like to develop techniques that support quick schema evolution without giving up on any of the other desirable features.

We address performance challenge with a *storage layer* to implement a practical means of actually storing the data. Conventionally, database systems have been designed with the goal of optimizing query processing. However, schema modifications (e.g., ALTER TABLE) are often time-consuming, heavy-weight operations in current systems. We utilize a vertically partitioned format for the storage layer. Our goal is to significantly reduce the performance penalty incurred due to schema modifications at a very modest cost of overhead in query processing.

Understanding Schema Evolution: When a schema has evolved over an extended period of time, it is difficult for a user to keep track of the changes. A natural need is to concisely convey to the user *how* a database has been evolving. For example, the user may query the relationship between columns in the initial schema and the final schema and how the transformation from old columns to new ones took place over time. We want to show users the gradual organic changes rather than a sudden transformation. We could keep track of all the changes step by step, which requires all changes to be maintained. If such information is not available, which is frequently the case when the user looks at external data sources, we seek to automatically discover such evolution from the data. Challenges involve mining conceptual changes from large amounts of changes to the database (e.g. Inferring the splitting of every "Name" column in each table to two "First Name" and "Last Name" columns, followed by a normalization of the names into a single table). Mining such inferences can be done using either just the data, or a combination of the data and provenance information.

4 Related Work

Database usability started to receive attention more than 25 years ago (23) and gained more momentum lately (36). Research in database usability has been mainly focusing on innovative and effective query interface design, including visual, text (i.e., keyword), natural language interfaces, direct manipulation interfaces, and spreadsheet interfaces.

Visual Interfaces: Query By Example (79), which is the first study on building a query interface not based on a database query language, allows users to implicitly construct queries by identifying examples of desired data elements. This work is followed more recently by QBT (65), Kaleidoquery (55), VISIONARY (9), MIX (54), Xing (27), and XQBE (13). Alternatively, forms-based query interface design has also been receiving attention. Early works on such interfaces include (26; 20), which provide users with visual tools to frame queries and to perform tasks such as database design and view definition. This direction is more recently followed by GRIDS (64) and Acuity (68),

and, in XML database systems, by FoXQ (1), EquiX (21), QURSED (60). Adaptive form construction is studied in DRIVE (53), which enables runtime context-sensitive interface editing for object-oriented databases, and in (38), which studies how forms can be automatically designed and constructed based on past query history. Recent work by Jayapandian and Jagadish proposes techniques for automatic construction of forms based on database schema and data (39) and expressive form customization (40).

Text Interfaces: The success of Information Retrieval (IR) style (i.e., keyword based) search among ordinary users has prompted database researcher to study a similar search interface for database systems. The goal is to maintain the simplicity of the search and exploit not only the textual content of the tuples, but also the structures within and across tuples to rank the results in a way that is more effective than the traditional IR-style ranking mechanism. For relational databases, this approach is first studied by Goldman et. al. in (28) and followed by many systems, including DBXplorer (2), BANKS (10), DISCOVER (34), and ObjectRank (7). For XML databases, the inherently more complicated structure within the database content allows the researchers to explore query languages ranging from pure keywords and approximate structural query, and has led to various projects including XSEarch (22), XRANK (29), JuruXML (16), FlexPath (5), Schema-Free XQuery (48), and Meaningful Summary Query (77). A more recent trend in keyword-based search is to analyze a keyword query and automatically discover the hidden semantic structures that the query carries. This trend has influenced the design of projects for both traditional database search (41) and web search (51).

Natural Language Interfaces: Constructing a natural language interface to databases has a long history (6). In particular, (66) analyzed the expressive power of a declarative query language (SEQUEL) in comparison to natural language. Most recently, NaLIX (47) proposed a generic natural language interface to XML database, which is capable of adapting to multiple domains through user feedbacks. However, to this day, natural language understanding is still an extremely difficult problem, and current systems tend to be unreliable or unable to answer questions outside a few predefined narrow domains (61).

Direct Manipulation Interfaces: Direct manipulation (67), although a crucial concept in the user interface field, is seldom mentioned in database literature. Pasta-3 (46) is one of the earliest efforts attempting a direct manipulation interface for databases, but its support of direct manipulation is limited to allowing users to manipulate *a query expression* with clicks and drags. Tioga-2 (4) (later developed into DataSplash (58)) is a direct manipulation database visualization tool, and its visual query language allows specification with a drag-and-drop interface. Its emphasis, however, is on visualization instead of querying. Recent work by Liu and Jagadish (50) develops a direct manipulation query interface based on an spreadsheet algebra.

Spreadsheet Interface: Spreadsheets have proven to be one of the most user-friendly and popular interfaces for handling data, partially evidenced by the ubiquity of Microsoft Excel. FOCUS (71) provides an interface for manipulating local tables. Its query operations are quite simple (e.g., allowing only one level of grouping and being highly

restrictive on the form of query conditions). Tableau (30), which is built on VizQL (31), specializes in interactive data visualization and is limited in querying capability. Spreadsheets have also been used for data cleaning (63), logic programming (70), visualization exploration (37), and photo management (43). Witkowski et al (75) proposed SQL extensions supporting spreadsheet-like computations in RDBMS.

Query interface is just one aspect of database usability, there are many other research fields that have direct or indirect impacts on the usability of databases, which we briefly describe below.

Personalization: Studies in this field attempt to customize database systems for each individual user and therefore making them easier to explore and extract information by the particular user, e.g., (24). In addition, studies have also been focusing on analyzing past query workloads to detect the user interests and provide better results tuned to those interests, e.g., (45; 19; 35). It is also worth noting that the notion of personalization has also found interest in the information retrieval community, where the ranking of search results is biased using a certain personalized metric (33; 42).

Automatic Database Management: To alleviate the burden on database administrators, commercial database systems come with a suite of auxiliary tools. The AutoAdmin project (3; 18) at Microsoft, initiated by Surajit Chaudhuri and his colleagues, makes great strides with respect to many aspects of database configuration including physical design and index tuning. Similarly, the Autonomic Computing project (49; 52) at IBM provides a platform to tune a database system, including query optimization. However, none of these projects deal with the user-level database usability that is the focus of this proposal.

Database Schema Design: This has been studied extensively (11; 76; 12; 59). There is a great deal of work on defining a good schema, both from the perspective of capturing real-life requirements (e.g., normalization) and supporting efficient queries. However, schema design has typically been considered a heavyweight, one-time operation, which is done by a technically skilled database administrator, based on careful requirements analysis and planning. The new challenge of enabling non-expert user to "give birth" to a database schema was posed recently (36), but no solution was provided.

Usability Study in Other Systems: Usability of information retrieval systems was studied in (72; 78), which analyzed usability errors and design flaws, and also in (25), which performed a comparison of usability testing methods. Principles of user-centered design were introduced in (44; 74), including how they could complement software engineering techniques to create interactive systems. Incorporating usability into the evaluation of computer systems was first studied in (14). An extensive user study was performed in (17) to identify the reasons for user frustration in computing experiences, while (15) takes a more formal approach to model user behavior for usability analysis. There is also a recent move in the software systems community to conduct serious user studies (69). However, for database systems in particular, these only scratch the surface of what needs to be done to improve usability.

References

[1] Abraham, R.: FoXQ - XQuery by forms. In: IEEE Symposium on Human Centric Computing Languages and Environments (2003)

[2] Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: A System for Keyword-Based Search over Relational Databases. In: ICDE (2002)

[3] Agrawal, S., Chaudhuri, S., Kollar, L., Marathe, A., Narasayya, V., Syamala, M.: Database Tuning Advisor for Microsoft SQL Server 2005. In: VLDB (2004)

[4] Aiken, A., Chen, J., Stonebraker, M., Woodruff, A.: Tioga-2: A direct manipulation database visualization environment. In: ICDE, pp. 208–217 (1996)

[5] Amer-Yahia, S., Lakshmanan, L.V.S., Pandit, S.: FleXPath: Flexible Structure and Full-Text Querying for XML. In: SIGMOD (2004)

[6] Androutsopoulos, I., Ritchie, G., Thanisch, P.: Natural Language Interfaces to Databases–an introduction. Journal of Language Engineering 1(1), 29–81 (1995)

[7] Balmin, A., Hristidis, V., Papakonstantinou, Y.: ObjectRank: Authority-Based Keyword Search in Databases. In: VLDB (2004)

[8] Bell, G., Gemmell, J.: A Digital Life (2007)

[9] Benzi, F., Maio, D., Rizzi, S.: Visionary: A Viewpoint-based Visual Language for Querying Relational Databases. Journal of Visual Languages and Computing 10(2) (1999)

[10] Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword Searching and Browsing in Databases using BANKS. In: ICDE (2002)

[11] Biskup, J.: Achievements of Relational Database Schema Design Theory Revisited. In: Thalheim, B. (ed.) Semantics in Databases 1995. LNCS, vol. 1358, pp. 29–54. Springer, Heidelberg (1998)

[12] Biskup, J.: Achievements of Relational Database Schema Design Theory Revisited. In: Thalheim, B. (ed.) Semantics in Databases 1995. LNCS, vol. 1358, pp. 29–54. Springer, Heidelberg (1998)

[13] Braga, D., Campi, A., Ceri, S.: *XQBE* (*XQ*uery *B*y *E*xample): A Visual Interface to the Standard XML Query Language. ACM Trans. Database Syst. 30(2) (2005)

[14] Brown, A.B., Chung, L.C., Patterson, D.A.: Including the Human Factor in Dependability Benchmarks. In: DSN Workshop on Dependability Benchmarking (2002)

[15] Butterworth, R., Blandford, A., Duke, D.: Using Formal Models to Explore Display-Based Usability Issues. Journal of Visual Languages and Computing 10(5) (1999)

[16] Carmel, D., Maarek, Y.S., Mandelbrod, M., Mass, Y., Soffer, A.: Searching XML Documents via XML Fragments. In: SIGIR (2003)

[17] Ceaparu, I., Lazar, J., Bessiere, K., Robinson, J., Shneiderman, B.: Determining Causes and Severity of End-User Frustration. International Journal of Human Computer Interaction 17(3) (2004)

[18] Chaudhuri, S., Weikum, G.: Rethinking Database System Architecture: Towards a Self-Tuning, RISC-style Database System. In: VLDB (2000)

[19] Chen, Z., Li, T.: Addressing Diverse User Preferences in SQL-Query-Result Navigation. In: SIGMOD (2007)

[20] Choobineh, J., Mannino, M.V., Tseng, V.P.: A Form-Based Approach for Database Analysis and Design. CACM 35(2) (1992)

[21] Cohen, S., Kanza, Y., Kogan, Y., Sagiv, Y., Nutt, W., Serebrenik, A.: EquiX–A Search and Query Language for XML. JASIST 53(6) (2002)

[22] Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: A Semantic Search Engine for XML. In: VLDB (2003)

[23] Date, C.J.: Database Usability. In: SIGMOD. ACM Press, New York (1983)
[24] Dong, X., Halevy, A.: A Platform for Personal Information Management and Integration. In: CIDR (2005)
[25] Doubleday, A., Ryan, M., Springett, M., Sutcliffe, A.: A Comparison of Usability Techniques for Evaluating Design. In: DIS (1997)
[26] Embley, D.W.: NFQL: The Natural Forms Query Language. ACM Trans. Database Syst. (1989)
[27] Erwig, M.: A Visual Language for XML. In: IEEE Symposium on Visual Languages,
[28] Goldman, R., Shivakumar, N., Venkatasubramanian, S., Garcia-Molina, H.: Proximity Search in Databases. In: VLDB (1998)
[29] Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: Ranked Keyword Search over XML Documents. In: SIGMOD (2003)
[30] Hanrahan, P.: VizQL: A Language for Query, Analysis and Visualization. In: SIGMOD, pp. 721–721 (2006)
[31] Hanrahan, P.: Vizql: a language for query, analysis and visualization. In: SIGMOD, p. 721 (2006)
[32] Hara, C., Davidson, S.: Reasoning about nested functional dependencies. In: PODS (1999)
[33] Haveliwala, T.: Topic-Sensitive PageRank: A Context-Sensitive Ranking Algorithm for Web Search. IEEE Transactions on Knowledge and Data Engineering 15(4), 784–796 (2003)
[34] Hristidis, V., Papakonstantinou, Y.: DISCOVER: Keyword Search in Relational Databases. In: VLDB (2002)
[35] Ioannidis, Y.E., Viglas, S.: Conversational Querying. Inf. Syst. 31(1), 33–56 (2006)
[36] Jagadish, H.V., Chapman, A., Elkiss, A., Jayapandian, M., Li, Y., Nandi, A., Yu, C.: Making database systems usable. In: SIGMOD (2007)
[37] Jankun-Kelly, T.J., Ma, K.-L.: A spreadsheet interface for visualization exploration. In: IEEE Visualization, pp. 69–76 (2000)
[38] Jayapandian, M., Jagadish, H.V.: Automating the Design and Construction of Query Forms. In: ICDE (2006)
[39] Jayapandian, M., Jagadish, H.V.: Automated creation of a forms-based database query interface. In: VLDB (2008)
[40] Jayapandian, M., Jagadish, H.V.: Expressive query specification through form customization. In: EDBT (2008)
[41] Jayram, T.S., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., Zhu, H.: Avatar Information Extraction System. IEEE Data Eng. Bull. 29(1), 40–48 (2006)
[42] Jeh, G., Widom, J.: Scaling Personalized Web Search. In: WWW, pp. 271–279 (2003)
[43] Kandel, S., Paepcke, A., Theobald, M., Garcia-Molina, H.: The photospread query language. Technical report, Stanford Univ. (2007)
[44] Kelley, J.F.: An Iterative Design Methodology for User-Friendly Natural Language Office Information Applications. ACM Trans. Database Syst. 2(1) (1984)
[45] Koutrika, G., Ioannidis, Y.: Personalization of Queries in Database Systems. In: ICDE (2004)
[46] Kuntz, M., Melchert, R.: Pasta-3's graphical query language: Direct manipulation, cooperative queries, full expressive power. In: VLDB, pp. 97–105 (1989)
[47] Li, Y., Yang, H., Jagadish, H.V.: NaLIX: A Generic Natural Language Search Environment for XML Data. ACM Transactions on Database Systems-TODS 32(4) (2007)
[48] Li, Y., Yu, C., Jagadish, H.V.: Enabling Schema-Free XQuery with Meaningful Query Focus. VLDB Journal (in press)

[49] Lightstone, S., Lohman, G.M., Haas, P.J., et al.: Making DB2 Products Self-Managing: Strategies and Experiences. IEEE Data Eng. Bull. 29(3), 16–23 (2006)
[50] Liu, B., Jagadish, H.V.: A spreadsheet algebra for a direct data manipulation query interface. In: ICDE (2009)
[51] Madhavan, J., Jeffery, S., Cohen, S., Dong, X., Ko, D., Yu, C., Halevy, A.: Web-scale Data Integration: You Can Only Afford to Pay As You Go. In: CIDR (2007)
[52] Markl, V., Lohman, G.M., Raman, V.: LEO: An Autonomic Query Optimizer for DB2. IBM Systems Journal 42(1), 98–106 (2003)
[53] Mitchell, K., Kennedy, J.: DRIVE: An Environment for the Organized Construction of User-Interfaces to Databases. In: Interfaces to Databases, IDS-3 (1996)
[54] Mukhopadhyay, P., Papakonstantinou, Y.: Mixing Querying and Navigation in MIX. In: ICDE (2002)
[55] Murray, N., Paton, N., Goble, C.: Kaleidoquery: A Visual Query Language for Object Databases. In: Advanced Visual Interfaces (1998)
[56] Nandi, A., Jagadish, H.V.: Assisted Querying using Instant-Response Interfaces. In: SIGMOD (2007)
[57] Nandi, A., Jagadish, H.V.: Qunits: queried units for database search. In: CIDR (2009)
[58] Olston, C., Woodruff, A., Aiken, A., Chu, M., Ercegovac, V., Lin, M., Spalding, M., Stonebraker, M.: Datasplash. In: SIGMOD, pp. 550–552 (1998)
[59] Papadomanolakis, E., Ailamaki, A.: Autopart: Automating schema design for large scientific databases using data partitioning. In: SSDBM (2004)
[60] Papakonstantinou, Y., Petropoulos, M., Vassalos, V.: QURSED: Querying and Reporting Semistructured Data. In: SIGMOD (2002)
[61] Popescu, A.-M., Etzioni, O., Kautz, H.A.: Towards a Theory of Natural Language Interfaces to Databases. In: IUI (2003)
[62] Qian, L., LeFevre, K., Jagadish, H.V.: Crius: User-friendly database design. PVLDB 4(2), 81–92 (2010)
[63] Raman, V., Hellerstein, J.M.: Potter's wheel: An interactive data cleaning system. In: VLDB, pp. 381–390 (2001)
[64] Sabin, R.E., Yap, T.K.: Integrating Information Retrieval Techniques with Traditional DB Methods in a Web-Based Database Browser. In: SAC (1998)
[65] Sengupta, A., Dillon, A.: Query by Templates: A Generalized Approach for Visual Query Formulation for Text Dominated Databases. In: ADL (1997)
[66] Sheneiderman, B.: Improving the Human Factors Aspect of Database Interactions. ACM Trans. Database Syst. 3(4) (1978)
[67] Shneiderman, B.: The future of interactive systems and the emergence of direct manipulation. Behaviour & Information Technology 1(3), 237–256 (1982)
[68] Sinha, S., Bowers, K., Mamrak, S.A.: Accessing a Medical Database using WWW-Based User Interfaces. Technical report, Ohio State University (1998)
[69] Soules, C., Shah, S., Ganger, G.R., Noble, B.D.: It's Time to Bite the User Study Bullet. Technical report, University of Michigan (2007)
[70] Spenke, M., Beilken, C.: A spreadsheet interface for logic programming. In: CHI, pp. 75–80 (1989)
[71] Spenke, M., Beilken, C., Berlage, T.: Focus: The interactive table for product comparison and selection. In: UIST, pp. 41–50 (1996)
[72] Sutcliffe, A., Ryan, M., Doubleday, A., Springett, M.: Model Mismatch Analysis: Towards a Deeper Explanation of Users' Usability Problems. Behavior & Information Technology 19(1) (2000)

[73] Tan, B., Peng, F.: Unsupervised query segmentation using generative language models and wikipedia. In: WWW (2008)
[74] Wasserman, A.I.: User Software Engineering and the Design of Interactive Systems. In: ICSE. IEEE Press, Piscataway (1981)
[75] Witkowski, A., Bellamkonda, S., Bozkaya, T., Dorman, G., Folkert, N., Gupta, A., Sheng, L., Subramanian, S.: Spreadsheets in rdbms for olap. In: SIGMOD (2003)
[76] Wong, S.K.M., Butz, C.J., Xiang, Y.: Automated database schema design using mined data dependencies. Journal of the American Society for Information Science 49, 455–470 (1998)
[77] Yu, C., Jagadish, H.V.: Querying Complex Structured Databases. In: VLDB (2007)
[78] Yuan, W.: End-User Searching Behavior in Information Retrieval: A Longitudinal Study. JASIST 48(3) (1997)
[79] Zloof, M.M.: Query-by-Example: the Invocation and Definition of Tables and Forms. In: VLDB (1975)