Organic Databases

H. V. Jagadish

Department of Computer Science and Engineering,
University of Michigan,
Ann Arbor, MI, USA
E-mail: jag@umich.edu

Li Qian

Department of Computer Science and Engineering,
University of Michigan,
Ann Arbor, MI, USA
E-mail: eql@umich.edu

Arnab Nandi

Department of Computer Science and Engineering,
Ohio State University,
Columbus, OH, USA
E-mail: arnab@cse.ohio-state.edu

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.

Indeed, the deliberate design model complicates not only database creation, but also database transformation (i.e., schema mapping). Because traditional schema mapping tasks are carefully engineered with declarative specification hidden beneath complex user interface. In this paper, we also study the issue of organic database transformation, which automatically induces schema mappings from sample target database instances.

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 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.

The emergence of "big data" in enterprise settings has also presented a unique set of critical data management challenges. Due to the sheer size, data is typically stored in large, unindexed data warehouses running on large clusters. Data is curated in a highly collaborative manner using data pipelines built by hundreds, if not thousands of engineers. Data manipulations are not restricted to simple database querying. They involve tasks such as information extraction and building of statistical models. Datasets range from completely unstructured to fully structured, and represent a wide variety of data models. While existing database systems pay a lot of attention to aspects such as query execution for simple database queries, the ability to deal with this new kind of data paradigm is still extremely primitive. For example, practical tasks such as "Which I.P.-address to zip-code table is most accurate to use while building a classifier for my user location data?" are encountered regularly and are currently solved by trial and error along with by duplication of engineer effort and cluster usage. Clearly, there are a host of data management issues, ranging from schema, workflow and provenance management to efficient indexing of heterogeneous structured data. These issues are permeate across all types of enterprise-class data, be it scientific or web-centric data management. With massive changes in the scale, size and complexity of both data and its use, a wide variety of research problems emerge.

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 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 [58] 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.

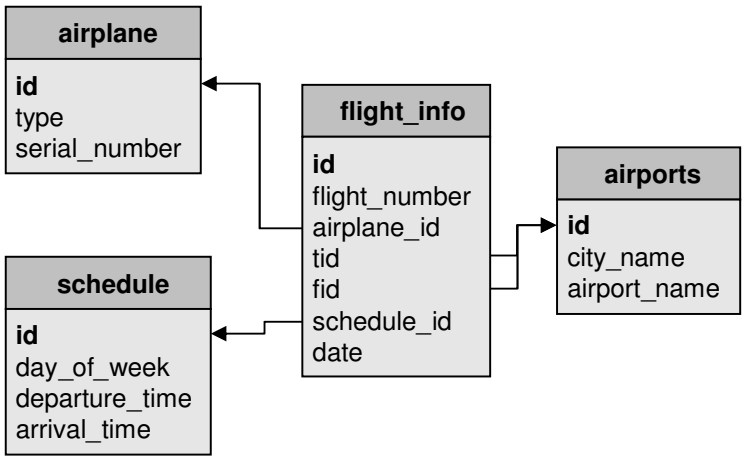


Figure 1 The base tables needed to store a "flight". A flight contains from location, destination, airplane info and schedule, yet consists of at least four tables. Note that an actual schema for such data is likely to involve many more attributes and tables.

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 2.1 *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.

2.4 Schema Mapping Challenge

Schema mapping has long been one of the most important problems in industry. Moreover, as the amount of structured Web-based information explodes, users are directly exposed to the task of combining, structuring and re-purposing information [15]. Unfortunately, schema mapping is extremely sophisticated with the state-of-the-art approaches.

Nowadays, schema mapping requires a substantial amount of careful planning in advance. These planning are usually based on complicated data transformation specification languages such as datalog and source-to-target tgds [28], which are difficult for normal users to understand. Furthermore, planning the mappings with modern tools requires declarative precision, which the users may not possess before they fully interpret the semantics of the mappings they are trying to construct.

Example 2.2 *When exploring the Yahoo Movies database, a user wishes to store the movie title as* **Name** *and the director name as* **Director** *in a target* **MyMovieInfo** *relation, as shown*[1] *in Figure 2. Given the fact that most users are unable/unwilling to specify the mappings in complex mapping languages, modern mapping tools usually offer a mapping interface as shown in Figure 3. However, users still have to precisely plan for two essential mapping components.*

First, similar to data location challenge described before, it can be potentially difficult to search for the source schema and locate the specific attribute that is being mapped to the corresponding target attribute. In a movie database, there are typically dozens of relations with hundreds of attributes. Worse, end-users usually have no access to foreign key constraints and/or database creation statements. In such a case, picking the right corresponding attribute can be a large pain.

Even if the set of correspondences are all correctly and precisely established, the user must face the structural specification challenge. In other words, how are these source attributes joined and projected to the target? Again, join prevents the user from linking desired concrete target concept to normalized stored information, and thus needs to be specified precisely. What if there are hidden intermediate join relations? What are the join attributes? All these planning questions are left to users who do not need to know these answers.

3 Proposed Solution

3.1 Presentation Data Model

We propose the use of a *presentation* data model [37], 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 [59], 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.

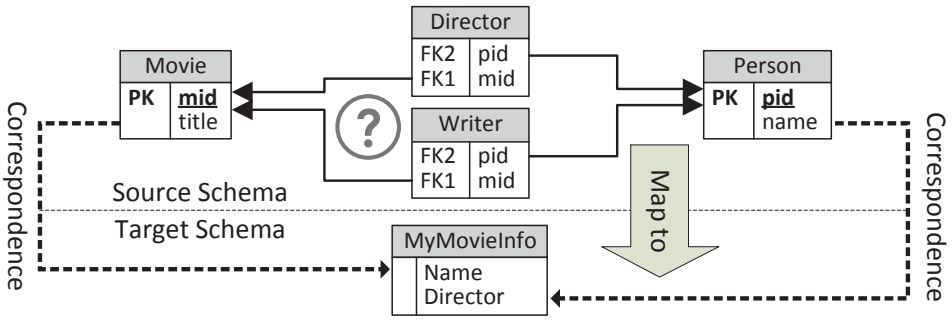


Figure 2 An Example Schema Mapping with The Question Mark Indicating a Join Path Ambiguity.

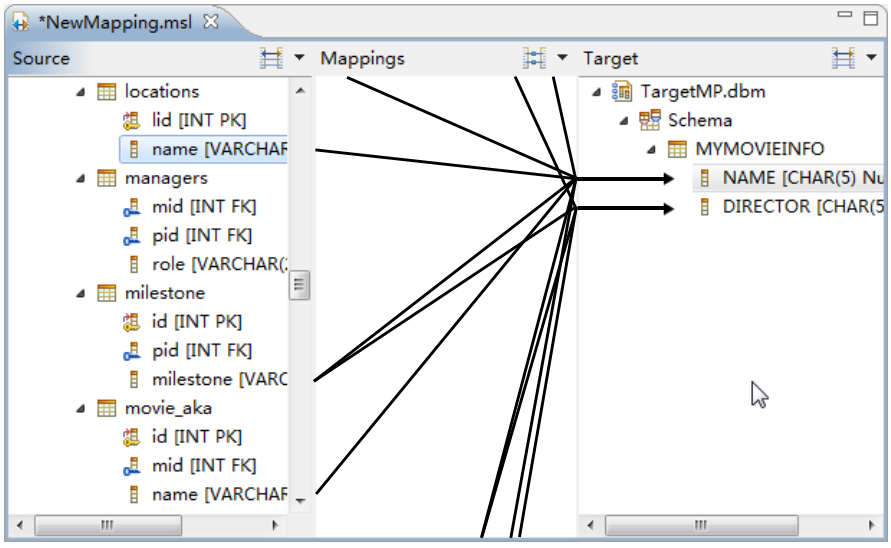
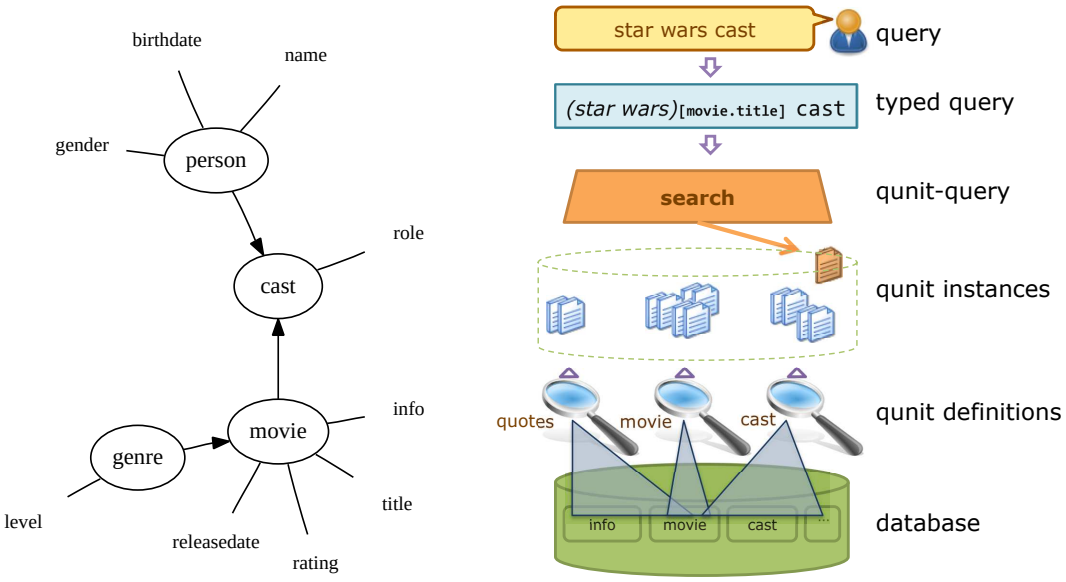


Figure 3 A Screenshot of IBM InfoSphere Data Architect



(a) An Simplified database schema (b) Qunit Search on IMDb

Figure 4 Qunit Example

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 4 (a) shows a simplified example schema of a movie database, which 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 4 (b). Queries are first processed to identify entities using standard query segmentation techniques [75]. 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).

A significant sub-challenge is the automated derivation of qunit definitions themselves. In addition to clustering-based techniques [79] that leverage the data and structure of the database, there is also a wealth of useful information that exists in the form of *external evidence*. External evidence can be in the form of existing "reports" – published results of queries to the database, or relevant web pages that present parts of the data. Such evidence is common in settings where such reports and web pages may be published and exchanged but the queries to the data are not published. Information extraction techniques such as wrapper induction [48] allow us to extract templates by considering each piece of evidence as a qunit instance, which an then be assembled into qunit definitions by mapping them to the database.

3.3 Addressing the 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 5 shows how Jane can organically grow the schema of the shopping list table. Initially, she has only columns for items to shop (Figure 5 (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 5 (b)). This makes the name attribute a level up; the rest of the columns forms a sub-relation "Gift" (shown in Figure 5 (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 5 (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.

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 5 is challenging using existing tools. For example, using conventional spreadsheet software, it is impossible to arrive at a hierarchical schema as shown in Figure 5 (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 5 (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 5 (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

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

Figure 5 Organic Schema Evolution

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 [33]. 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 [64].

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.

3.4 Addressing the Schema Mapping Challenge

In this section, we address the schema mapping challenge (Sec. 2.4) by proposing a sample-driven mapping approach, as opposed to the traditional match-driven mapping approach adopted by most of the state-of-the-art schema mapping tools. The sample-driven mapping approach allows the user to freely provide sample target instances for the system to automatically infer the desired schema mapping, enabling organic data transformation by reducing user-side planning burden.

Let us examine a natural extension of Example 2.2, in which the user wishes to establish the mapping from the whole source to the toy target consists of the movie title, the director's name and the production company. Even with such a simple target, the traditional mapping approach could imply a great planning pain. First, for a given target attribute, there may be several possible candidate source attributes that can match. Moreover, even if the source attributes have a one-to-one matching with the target attributes, there may be various ways these source attributes can be joined. To resolve either ambiguity with a match-driven interface may be potentially overwhelming for the user, as shown in the top part of Figure 6.

With the sample-driven approach, the user can bypass the source-to-target-correspondence establishment and join path construction. All she needs to do is to type in sample instances in a restful WYSIWYG target relation, as shown in the bottom part in Figure 6. For instance, the user begins by entering "Avatar", and the system may find the value in both source attributes *Movie.title* and *Company.name*. As the user enters more movie names, such as "Harry Potter", the set of attributes eventually converges to a single attribute *Movie.title*. The same concept applies to join path specification. After the user inputs "Avatar" and "James Cameron" in the first row, the relation between these two entries are not clear enough. It may stand for a movie-director relationship or a movie-writer relationship. However, after the user enters "Harry Potter" and "David Yates", the system knows the desired relationship must be movie-director, since the writer of "Harry Potter" is "J.K. Rowling". By this means, the user easily specifies the mapping without planning for the underlying complicated details.

In the remainder of this section, we briefly highlight the challenges in supporting such organic schema mapping system and our solutions.

Mapping Deduction: The key challenge behind the scene is to automatically deduce the desired mapping using just the user-provided sample target instances. This is essentially difficult because of two reasons. First, the user input is small and incomplete compared with the whole desired target relation which the user eventually wishes to generate. Second, there is no prior-knowledge of what this "desired target" would be. In other words, there may be a large number of candidate mappings that are consistent with the current user input, yet no one knows which one is the ground truth. Therefore, we have to keep track of all possible mappings.

Indeed, the deduction problem is a reverse-engineering problem which uses only incomplete information to search for the most suitable process that would generate such information, while the number of possible processes may be many. The reason for such ambiguity, to be viewed from the schema mapping context, is obvious. For each user-input data, there may be multiple source attributes which contain that piece of data. Moreover, for each pair of source attributes that are being matched to the target, there may be multiple ways to join them. Consequently, the critical question to the sample-driven mapping approach becomes how to effectively compute the set of candidate mappings that would yield the current user-input target sample instances.

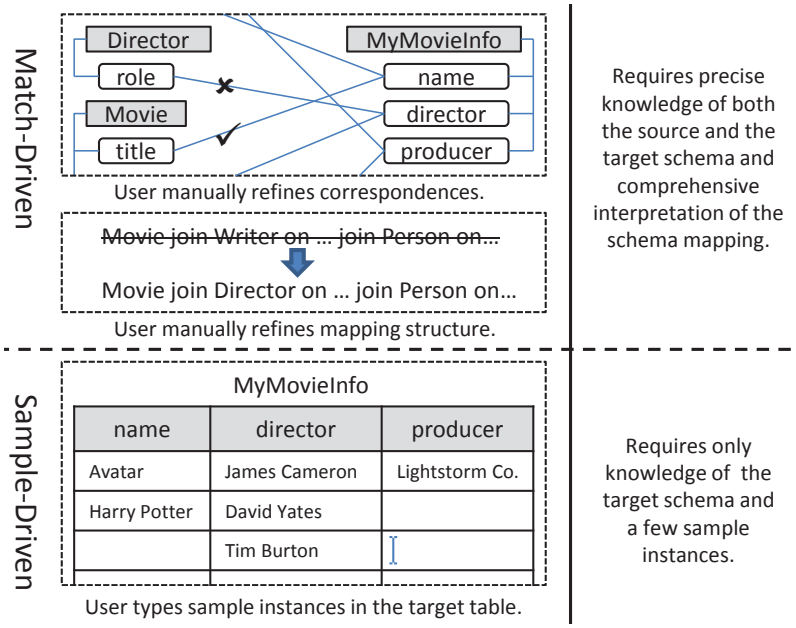


Figure 6 Traditional Schema Mapping v.s. Organic Schema Mapping with Sample-Driven Approach

In traditional schema mapping design, database engineers may spend sufficient time on discussing and choosing the optimal mapping. In the organic mapping scenario, however, these mappings have to be generated quickly enough so that the user can obtain "interactive-speed" feedback, allowing her to review the current system status before continuing to provide more samples or stopping if the system has generated the desired mapping. As a result, the system has to generate the set of candidate mappings with satisfying performance.

We address this mapping deduction challenge by abstracting schema mappings into mapping paths, and developing algorithms that efficiently derive bigger mapping paths from smaller ones, where the smallest mapping paths are generated by depth-limited breadth-first-searching user-input connections in the source database instances. The highlight of the algorithm is, every piece of required information is captured in those smallest mapping paths, and the process of constructing them into the final mapping candidates follows a natural bottom-up procedure with a "weaving" operation that can be completely done in memory. It is proved that the algorithm is sound and complete. And because all database accesses only occur when generating the smallest mapping paths, the overall performance is noticeable high and is able to meet practical requirements.

Mapping Pruning:

After we deduct the set of all mapping candidates, we have to efficiently prune them to obtain the final desired mapping. This process is also time-critical since it has to be done when the system interacts with continuous user-input. Here we propose two approaches for mapping pruning.

After the initial set of candidate mappings is generated, the user may continue to enter additional samples to prune the candidate set. This basic static pruning functions in two steps. In the first step, source attributes in existing candidate mappings are being checked towards newly input samples. Those source attributes not containing the new samples, together with the corresponding mappings will be pruned. In the second step, the connections between newly input samples are being checked against candidate mappings and mappings that do not satisfy the new connections will be removed.

In complement to the static pruning, we propose dynamic pruning based on automatic sample instance recommendation. In general, a user may not realize the critical differences between various candidate mappings and

continues with samples supporting a single existing mapping. This, indeed, renders the candidate set convergence time not upper-bounded. In order to make the candidate set converge faster, we can automatically construct target instances that satisfy only part of the candidate set, and ask the user if such a selection is desired. By this means, the size of the candidate set quickly shrinks and the user is able to obtain the goal mapping within just a few interactions.

4 Related Work

Database usability started to receive attention more than 25 years ago [23] and gained more momentum lately [37]. 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 [82], 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 [67], Kaleidoquery [57], VISIONARY [9], MIX [56], Xing [27], and XQBE [12]. 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 [66] and Acuity [70], and, in XML database systems, by FoXQ [1], EquiX [21], QURSED [62]. Adaptive form construction is studied in DRIVE [55], which enables runtime context-sensitive interface editing for object-oriented databases, and in [39], 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 [40] and expressive form customization [41].

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 [29] and followed by many systems, including DBXplorer [2], BANKS [10], DISCOVER [35], 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 [30], JuruXML [16], FlexPath [5], Schema-Free XQuery [50], and Meaningful Summary Query [80]. 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 [42] and web search [53].

Natural Language Interfaces: Constructing a natural language interface to databases has a long history [6]. In particular, [68] analyzed the expressive power of a declarative query language (SEQUEL) in comparison to natural language. Most recently, NaLIX [49] 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 [63].

Direct Manipulation Interfaces: Direct manipulation [69], although a crucial concept in the user interface field, is seldom mentioned in database literature. Pasta-3 [47] 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 [60]) 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 [52] 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 [73] 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 [31], which is built on VizQL [32], specializes in interactive data visualization and is limited in querying capability. Spreadsheets have also been used for data cleaning [65], logic programming [72], visualization exploration [38], and photo management [44]. Witkowski et al [77] 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., [46, 19, 36]. 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 [34, 43].

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 [51, 54] 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, 78, 61]. 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 [37], but no solution was provided.

Usability Study in Other Systems: Usability of information retrieval systems was studied in [74, 81], 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 [45, 76], including how they could complement software engineering techniques to create interactive systems. Incorporating usability into the evaluation of computer systems was first studied in [13]. An extensive user study was performed in [17] to identify the reasons for user frustration in computing experiences, while [14] 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 [71]. However, for database systems in particular, these only scratch the surface of what needs to be done to improve usability.

5 Conclusion

Today, many users have to manage their own data, without the luxury of having it managed for them by a trained DBA. Without technical training, it is difficult for ordinary users to reason about abstract concepts, such as schema, let alone successfully design a database schema or map the schema of an unfamiliar database. To meet the needs of such users, we have introduced the notion of an organic database C distinguished from the traditional carefully engineered database. In this paper, we showed two instantiations of this concept, one for creating a database and one for schema mapping in data integration.

Acknowledgement

The project is supported in part by NSF grants IIS 1017296 and IIS 1250880.

References

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

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

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

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

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

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

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

[8] G. Bell and J. Gemmell. A Digital Life, 2007.

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

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

[11] J. Biskup. Achievements of relational database schema design theory revisited. In *Semantics in Databases*, pages 29–54. Springer-Verlag, 1998.

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

[13] A. B. Brown, L. C. Chung, and D. A. Patterson. Including the Human Factor in Dependability Benchmarks. In *DSN Workshop on Dependability Benchmarking*, 2002.

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

[15] M. Cafarella, A. Halevy, and N. Khoussainova. Data integration for the relational web. *VLDB*, 2(1):1090–1101, 2009.

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

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

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

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

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

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

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

[23] C. J. Date. Database Usability. In *SIGMOD*, New York, NY, USA, 1983. ACM Press.

[24] X. Dong and A. Halevy. A Platform for Personal Information Management and Integration. In *CIDR*, 2005.

[25] A. Doubleday, M. Ryan, M. Springett, and A. Sutcliffe. A Comparison of Usability Techniques for Evaluating Design. In *DIS*, 1997.

[26] D. W. Embley. NFQL: The Natural Forms Query Language. *ACM Trans. Database Syst.*, 1989.

[27] M. Erwig. A Visual Language for XML. In *IEEE Symposium on Visual Languages*, 2000.

[28] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, May 2005.

[29] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. In *VLDB*, 1998.

[30] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD*, 2003.

[31] P. Hanrahan. VizQL: A Language for Query, Analysis and Visualization. *SIGMOD*, pages 721–721, 2006.

[32] P. Hanrahan. Vizql: a language for query, analysis and visualization. In *SIGMOD*, page 721, 2006.

[33] C. Hara and S. Davidson. Reasoning about nested functional dependencies. In *PODS*, 1999.

[34] T. Haveliwala. Topic-Sensitive PageRank: A Context-Sensitive Ranking Algorithm for Web Search. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):784–796, 2003.

[35] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *VLDB*, 2002.

[36] Y. E. Ioannidis and S. Viglas. Conversational Querying. *Inf. Syst*, 31(1):33–56, 2006.

[37] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, 2007.

[38] T. J. Jankun-Kelly and K.-L. Ma. A spreadsheet interface for visualization exploration. In *IEEE Visualization*, pages 69–76, 2000.

[39] M. Jayapandian and H. V. Jagadish. Automating the Design and Construction of Query Forms. In *ICDE*, 2006.

[40] M. Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. In *VLDB*, 2008.

[41] M. Jayapandian and H. V. Jagadish. Expressive query specification through form customization. In *EDBT*, 2008.

[42] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar Information Extraction System. *IEEE Data Eng. Bull.*, 29(1):40–48, 2006.

[43] G. Jeh and J. Widom. Scaling Personalized Web Search. *WWW*, pages 271–279, 2003.

[44] S. Kandel, A. Paepcke, M. Theobald, and H. Garcia-Molina. The photospread query language. Technical report, Stanford Univ., 2007.

[45] J. F. Kelley. An Iterative Design Methodology for User-Friendly Natural Language Office Information Applications. *ACM Trans. Database Syst.*, 2(1), 1984.

[46] G. Koutrika and Y. Ioannidis. Personalization of Queries in Database Systems. In *ICDE*, 2004.

[47] M. Kuntz and R. Melchert. Pasta-3's graphical query language: Direct manipulation, cooperative queries, full expressive power. In *VLDB*, pages 97–105, 1989.

[48] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1):15–68, 2000.

[49] Y. Li, H. Yang, and H. V. Jagadish. NaLIX: A Generic Natural Language Search Environment for XML Data. *ACM Transactions on Database Systems-TODS*, 32(4), 2007.

[50] Y. Li, C. Yu, and H. V. Jagadish. Enabling Schema-Free XQuery with Meaningful Query Focus. *VLDB Journal*, in press.

[51] S. Lightstone, G. M. Lohman, P. J. Haas, et al. Making DB2 Products Self-Managing: Strategies and Experiences. *IEEE Data Eng. Bull*, 29(3):16–23, 2006.

[52] B. Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *ICDE*, 2009.

[53] J. Madhavan, S. Jeffery, S. Cohen, X. Dong, D. Ko, C. Yu, and A. Halevy. Web-scale Data Integration: You Can Only Afford to Pay As You Go. In *CIDR*, 2007.

[54] V. Markl, G. M. Lohman, and V. Raman. LEO: An Autonomic Query Optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[55] K. Mitchell and J. Kennedy. DRIVE: An Environment for the Organized Construction of User-Interfaces to Databases. In *Interfaces to Databases (IDS-3)*, 1996.

[56] P. Mukhopadhyay and Y. Papakonstantinou. Mixing Querying and Navigation in MIX. In *ICDE*, 2002.

[57] N. Murray, N. Paton, and C. Goble. Kaleidoquery: A Visual Query Language for Object Databases. In *Advanced Visual Interfaces*, 1998.

[58] A. Nandi and H. V. Jagadish. Assisted Querying using Instant-Response Interfaces. In *SIGMOD*, 2007.

[59] A. Nandi and H. V. Jagadish. Qunits: queried units for database search. *CIDR*, 2009.

[60] C. Olston, A. Woodruff, A. Aiken, M. Chu, V. Ercegovac, M. Lin, M. Spalding, and M. Stonebraker. Datasplash. In *SIGMOD*, pages 550–552, 1998.

[61] E. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, 2004.

[62] Y. Papakonstantinou, M. Petropoulos, and V. Vassalos. QURSED: Querying and Reporting Semistructured Data. In *SIGMOD*, 2002.

[63] A.-M. Popescu, O. Etzioni, and H. A. Kautz. Towards a Theory of Natural Language Interfaces to Databases. In *IUI*, 2003.

[64] L. Qian, K. LeFevre, and H. V. Jagadish. Crius: User-friendly database design. *PVLDB*, 4(2):81–92, Nov. 2010.

[65] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[66] R. E. Sabin and T. K. Yap. Integrating Information Retrieval Techniques with Traditional DB Methods in a Web-Based Database Browser. In *SAC*, 1998.

[67] A. Sengupta and A. Dillon. Query by Templates: A Generalized Approach for Visual Query Formulation for Text Dominated Databases. In *ADL*, 1997.

[68] B. Sheneiderman. Improving the Human Factors Aspect of Database Interactions. *ACM Trans. Database Syst.*, 3(4), 1978.

[69] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour & Information Technology*, 1(3):237–256, 1982.

[70] S. Sinha, K. Bowers, and S. A. Mamrak. Accessing a Medical Database using WWW-Based User Interfaces. Technical report, Ohio State University, 1998.

[71] C. Soules, S. Shah, G. R. Ganger, and B. D. Noble. It's Time to Bite the User Study Bullet. Technical report, University of Michigan, 2007.

[72] M. Spenke and C. Beilken. A spreadsheet interface for logic programming. In *CHI*, pages 75–80, 1989.

[73] M. Spenke, C. Beilken, and T. Berlage. Focus: The interactive table for product comparison and selection. In *UIST*, pages 41–50, 1996.

[74] A. Sutcliffe, M. Ryan, A. Doubleday, and M. Springett. Model Mismatch Analysis: Towards a Deeper Explanation of Users' Usability Problems. *Behavior & Information Technology*, 19(1), 2000.

[75] B. Tan and F. Peng. Unsupervised query segmentation using generative language models and wikipedia. In *WWW*, 2008.

[76] A. I. Wasserman. User Software Engineering and the Design of Interactive Systems. In *ICSE*, Piscataway, NJ, USA, 1981. IEEE Press.

[77] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Sheng, and S. Subramanian. Spreadsheets in rdbms for olap. In *SIGMOD*, 2003.

[78] S. K. M. Wong, C. J. Butz, and Y. Xiang. Automated database schema design using mined data dependencies. *Journal of the American Society for Information Science*, 49:455–470, 1998.

[79] C. Yu and H. V. Jagadish. Schema Summarization. In *VLDB*, 2006.

[80] C. Yu and H. V. Jagadish. Querying Complex Structured Databases. In *VLDB*, 2007.

[81] W. Yuan. End-User Searching Behavior in Information Retrieval: A Longitudinal Study. *JASIST*, 48(3), 1997.

[82] M. M. Zloof. Query-by-Example: the Invocation and Definition of Tables and Forms. In *VLDB*, 1975.