Making Database Systems Usable

H. V. Jagadish Adriane Chapman Aaron Elkiss
Magesh Jayapandian Yunyao Li Arnab Nandi Cong Yu

{jag,apchapma,aelkiss,jmagesh,yunyaol,arnab,congy}@umich.edu

University of Michigan
Ann Arbor, MI 48109-2122

ABSTRACT

Database researchers have striven to improve the capability of a database in terms of both performance and functionality. We assert that the *usability* of a database is as important as its capability. In this paper, we study why database systems today are so difficult to use. We identify a set of five pain points and propose a research agenda to address these. In particular, we introduce a *presentation data model* and recommend *direct data manipulation* with a *schema later* approach. We also stress the importance of *provenance* and of *consistency* across presentation models.

Categories and Subject Descriptors

H.2.0 [**General**]; H.5.0 [**General**]

General Terms

Design, Human Factors

Keywords

Database, Usability, User Interface

1. INTRODUCTION

Database technology has made great strides in the past decades. Today, we are able to efficiently process ever larger numbers of ever more complex queries on ever more humongous data sets. As a field, 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. The simple keyword-based query mechanism allows web users to issue queries freely; the almost instantaneous response encourages the user to refine queries until satisfactory results are found. While search engines today are still far from perfect, their huge success suggests that *usability* is key. An information system provides value to its users through the ability to get information into and out of the system easily and efficiently. Unfortunately, databases today are hard to design, hard to modify, and hard to query.

One obvious question to ask is whether database systems can simply have a search engine sit on top of them and let the search engine handle the interaction with the users. The answer, we argue, is "No." While the search engine interface works well for the web, it does not address all the usability problems database systems are facing. This is due to many characteristics that stem from users' expectations for interacting with databases, which are fundamentally different from expectations for the web.

The first characteristic that users expect is the ability to query the database in a more sophisticated way. While users are content with searching the web with keywords, they want to express more complex query semantics when interacting with databases simply because they know databases are more than collections of documents and there are structures

Supported in part by NSF grant IIS 0438909 and NIH grants R01 LM008106 and U54 DA021519. We thank Mark Ackerman, Jignesh Patel and Barbara Mirel for their comments on a draft of this paper.

inside databases that can be used to answer their queries. The use of boolean connectives, quote phrases, and colon prefixes are just the beginning of the complex semantics we can expect from database users. For example, when querying an airline database for available flights, a user will naturally search on explicit travel dates, departure/arrival cities or airports, and sometimes even airlines. Such complex semantics cannot be handled by a keyword-based interface that search engines provide.

The second characteristic is that users expect more precise and complete answers from database search. One of the hidden reasons behind the success of search engines is the fact that web users can tolerate a search that returns many irrelevant results (i.e., less than perfect precision) and they typically do not care if there are relevant results missing (i.e., less than perfect recall), as long as *some* relevant results are returned to them. In the database world, however, these assumptions no longer hold. Users issuing a database search expect to obtain *all* and *only* the relevant results: anything less than perfect precision and recall will have to be explained to the user.

The third characteristic is that users have an expectation of structure in the result. In the case of a web search, a user expects simply a set of links, with almost no interrelationship between them. In the case of a database search, a user may expect to see a table, a network, a spatial presentation on a map, or a set of points in a multidimensional space — the specific structure depends on the users mental model of the application.

The fourth characteristic is that users often expect to create and update databases. While search engines are all about searching, database users frequently want to design new databases to store their own information or to generate information to put into existing databases. The usability issues in designing a database structure from scratch and creating structured information for existing databases are completely unexplored so far and need to be addressed for databases to become widely used by ordinary users.

These fundamental characteristics suggest that the database research community needs to think about database usability not just as a query interface, but as a more comprehensive and integral component of the database systems.

Structured query models like SQL and XQuery are the current means provided by database systems to allow users to express complex query semantics. While powerful, those models are fundamentally difficult for users to adopt because they require users to fully comprehend the structure of the database (i.e., schema) and to express queries in terms of that particular structure. We argue that while the logical schema of the database is an appropriate abstraction over the underlying physical schema for data organization, it is still at a level too low to serve as the abstraction with which users should interact directly. Instead, a higher level *presentation data model* abstraction is needed to allow users to structure information in a natural way. We describe this problem of "painful relations" in Section 4.1. Because different users have different views (i.e. presentation data models) on how information should be organized, one would naturally like those presentation data models to be personalized for individual users. Our experience with the MiMI [51] project, however, has told us a different story. When users are presented with multiple ways to access the information but do not understand the underlying differences between the views, they tend to become confused and lose trust in the system. A good presentation data model can be enormously helpful, but too many such models becomes counterproductive. We describe this problem of "painful options" in Section 4.2.

The expectation of perfect precision and recall for database search introduces yet another usability issue: the need to issue explanations to the user when the database system produces unexpected results or fails to produce the expected results. The first case is a failure in precision—some of the results produced are not relevant in the mind of the user. The database system will need to be prepared to answer questions like "where does this result come from?" and provenance tracking [89] is essential in this regard. Second, some potentially relevant results may not be produced—a failure in recall. Some of those missing results can be identified by the system. For example, consider a query asking for "all flights with booking fee less than $10" and assume there are flights in the airline database that do not have a known booking fee. It is reasonable, or even desirable, not to return those flights to the user. However, the system must be able to explain to the user that those flights are not included in the result because their booking fees are unknown, and not because their booking fees exceed $10. Some missing results cannot ever be identified by the system. For example, consider a query asking for "a Tuesday flight from DTW to PEK on any airline" and assume the airline database does not carry flights from Northwest Airlines, which does have a flight between DTW and PEK. While the system may not be aware of the existence of such a flight with Northwest Airlines, it still needs to explain to the user that this omission is due to its incomplete coverage. We describe this problem of "unexpected pain" in Section 4.3.

Another way in which a user can get unexpected results from a system is when the user makes an error in specifying the query. Unfortunately, given how difficult it can be to specify a query correctly, such errors are all too common. A fundamental difficulty in a traditionally architected database system is that the user has a labor-intensive query construction phase followed by a potentially lengthy query evaluation phase, so that after the results are obtained the costs to reformulating the query are often high. Worse still, errors in query construction remain uncaught until the construction phase is completed and the query submitted for evaluation. This sort of write and debug cycle is something we computer scientists get used to as part of learning how to program. Other users need mechanisms so that they can see what they are doing with the database at all times. This absence of the ability to manipulate data directly is what we call "unseen pain" and discussed in Section 4.4.

Finally, the users' expectation of being able to create a database from scratch and/or to create structured information for an existing database introduces a whole new set of usability issues that are currently unexplored. Requiring a user to go through a rigorous schema design process before any information can be stored, as in the case of many current database design practices, puts too much burden on the user and runs the risk of user forgoing a database approach altogether. Similarly, requiring a user to study the existing database schema and restructure their data according to this schema before she can update the database with her own data also imposes an unnecessary burden. We describe this problem of "birthing pain" in Section 4.5.

Paper Outline: The rest of the paper is organized as follows. We describe the current research activities related to database usability in Section 2 and describe our ongoing MiMI project [51] as a case study of database usability in Section 3. In Section 4, we describe in detail the usability challenges facing database systems. Section 5 presents a research agenda that suggests some directions for future research in database usability. Finally, we conclude in Section 6.

2. CURRENT APPROACHES

There is evidence that human error is the leading cause for the failure of complex systems [77, 15]. To this effect, the nature of human usage has received considerable attention in research, e.g., there is a recent move in the software systems community to conduct serious user studies [91]. Database usability started to receive attention more than 25 years ago [32]. Since then, research in database usability has been following two main directions: innovative query interface design (including both visual and keyword-based interfaces) and database personalization. We describe recent accomplishments in those areas, as well as other related areas like automatic configuration and management of database systems.

2.1 Visual Interface

Visual query specification interface is perhaps the oldest and most prominent field related to database usability, in term of both academic research (e.g., QBE [100]) and industrial practices (e.g., Microsoft Access and IBM Visual XQuery Builder). Many visual query interfaces have been proposed to assist users in building queries incrementally, including XQBE [13], MIX [71], Xing [37], VISIONARY [9], Kaleidoquery [73] and QBT [85].

Forms-based query interface design has also been receiving attention. Early works on such interfaces include [26, 36] and provide users with visual tools to frame queries and to perform tasks such as database design and view definition. The GRIDS system [84] generates forms that allow users to pose queries in a semi-IR, semi-declarative fashion; the Acuity project [90] developed form generation techniques for data-entry operations such as updating tables. More recently in XML database systems, efforts have been made to shield users from both the details of the XQuery syntax and the textual representation of XML. FoXQ [1] and EquiX [28] are systems that helps users build queries incrementally by navigating through layers of forms. Semi-automatic form generation tools have been proposed in QURSED [79]. Furthermore, [93] proposes the use of XML rather than HTML to represent forms, making them more reusable, scalable, machine-readable and easier to standardize. Another interesting project in UI design is DRIVE [68], a runtime user interface development environment for object-oriented databases, which uses the NOODL data model to enable context-sensitive interface editing.

2.2 Text Interface

Our sister field of information retrieval has had wider adoption by normal users. This has prompted database interface designers to take the approach of providing database systems with an IR-style keyword-based search interface.

Many systems such as DBXplorer [2], BANKS [11], DISCOVER [47] and an early work of Goldman et al. [40] attempt to extend the simplicity of keyword search to relational data. This is not merely an integration of full-text search with relational query capability—such an approach still requires knowledge of the database schema. Rather, the core principle is to provide keyword search *across* tuples. Most of these systems find tuples that individually match each keyword and then find a path of primary/foreign key relationships that relate the tuples. Result ranking is typically provided based on the path of primary/foreign key relationships. A common ranking approach is to use some variation of PageRank [14], where documents and hyperlinks are replaced with tuples and referential constraints.

A parallel thread of research examines keyword search in XML databases. The basic problem is the same as in relational databases: given a set of keywords, we must find data fragments that match the individual keywords. Instead of only referential constraints, XML databases mostly have parent/child relationships between individual elements. The problem of determining if a data fragment is meaningfully related becomes much more important. Approaches to determine the meaningfulness as well as the relevance of a data fragment have ranged from simple tree distance used by XSEarch [29] to XRANK's adaptation of PageRank [41] to approaches such as Schema-Free XQuery [62, 63] that look at either the entire database or the database schema [98] to determine if a data fragment is meaningful.

A different approach with a long history is the construction of natural language interfaces to databases [5]. Natural language understanding is an extremely difficult problem, and commercial systems such as Microsoft English Query [12] tend to be unreliable or unable to answer questions outside a manually predefined narrow domain [83].

Other systems assume the user has some imperfect knowledge of the structure of the data as could occur with heterogeneous or evolving schema. This is an intermediate step in user complexity between pure keyword search and rigid structural search. Research such as FleXPath [4] and the work of Kanza and Sagiv [54] has focused on relaxation of fully specified structural queries; other systems such as JuruXML [21] support querying and result ranking based on similarity to a user-specified XML fragment.

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 [52] as well as web search [65].

2.3 Context and Personalization

Advancements in query interface design, while making it easier for users to interact with the database, are mostly generic: they do not take into account the specific user and her unique problems. The notion of *personalization* addresses this problem by attempting to customize database systems for each individual user [33]. This approach has received great attention in the context of websites [80, 69], where the content and structure of the website is tailored to the needs of each user by analyzing usage patterns. The notion of user context and personalization has also found interest in the information retrieval community, where the ranking of search results is biased using a certain personalized metric [45, 46, 53].

Database research has made advancements in accommodating user and contextual information into query processing. Koutrika and Ioannidis [56] define a user preference model and describe methods to evaluate the degree of personal interest in query results. Chen and Li [24] provide methods to mine query logs and cluster results according to user characteristics. Ioannidis and Viglas [49] also propose the idea of conversational querying in which queries are interpreted in the context of the previous queries in a query session.

2.4 Other Related Work

Commercial database systems come with a suite of auxiliary tools. The AutoAdmin project [3, 23] at Microsoft, initiated by Surajit Chaudhury 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 [64, 66] 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 paper.

Much work has been done by the HCI community in the area of usability improvement for computer system interfaces in general. Some of the earliest works in database usability includes [87], which analyzed the expressive power of a declarative query language (SEQUEL) in comparison to natural language. Usability of information retrieval systems was studied in [92, 99], which analyzed usability errors and design flaws, and also in [34], which performed a comparison of usability testing methods. Principles of user-centered design were introduced in [55, 94], including how they could complement software engineering techniques to create interactive systems. Incorporating usability into the evaluation of computer systems was studied in [16], which analyzed human behavior with a dependability benchmark. An extensive user study was performed in [22] to identify the reasons for user frustration in computing experiences, while [20] takes a more formal approach to model user behavior for usability analysis. However, for database systems in particular, these only scratch the surface of what needs to be done to improve usability.

3. A CASE STUDY

Just a few short years ago, we were a traditional database research group at the University of Michigan, focusing on data structures, algorithms, and performance. Usability was not a topic that we paid too much attention to. A significant project at that time was the development of Timber [50], a native XML database. As we looked for challenging applications to run on our database system, we started collaborating with biologists. When we put their data on our system and had them try to use it, we became aware of many unexpected issues. The insights gleaned from watching very smart but mostly non-technical people use database systems, both Timber and commercial relational systems, led to the ideas presented in this paper. In this section, we present a short history of these efforts as essential context for what follows.

After the sequencing of the human genome was completed, biologists began focusing their attention on the proteins expressed by these genes, their interactions, and their functions. Scientists perform a wide variety of experiments to determine which proteins interact with one another. These experiments have varying degrees of reliability. Several public databases of protein interactions have arisen, each with its own focus (e.g., organism, disease, high-throughput technologies, etc.). Protein entries are sometimes repeated within and across the repositories. A scientist interested in learning about a particular protein might have to visit half a dozen sites and merge information obtained from them, some overlapping, some even contradictory. We created MiMI [51], a deep integration of several of the best-regarded protein interaction databases. Provenance was retained to describe where the data originated, and the entire dataset and metadata were stored in Timber [50].

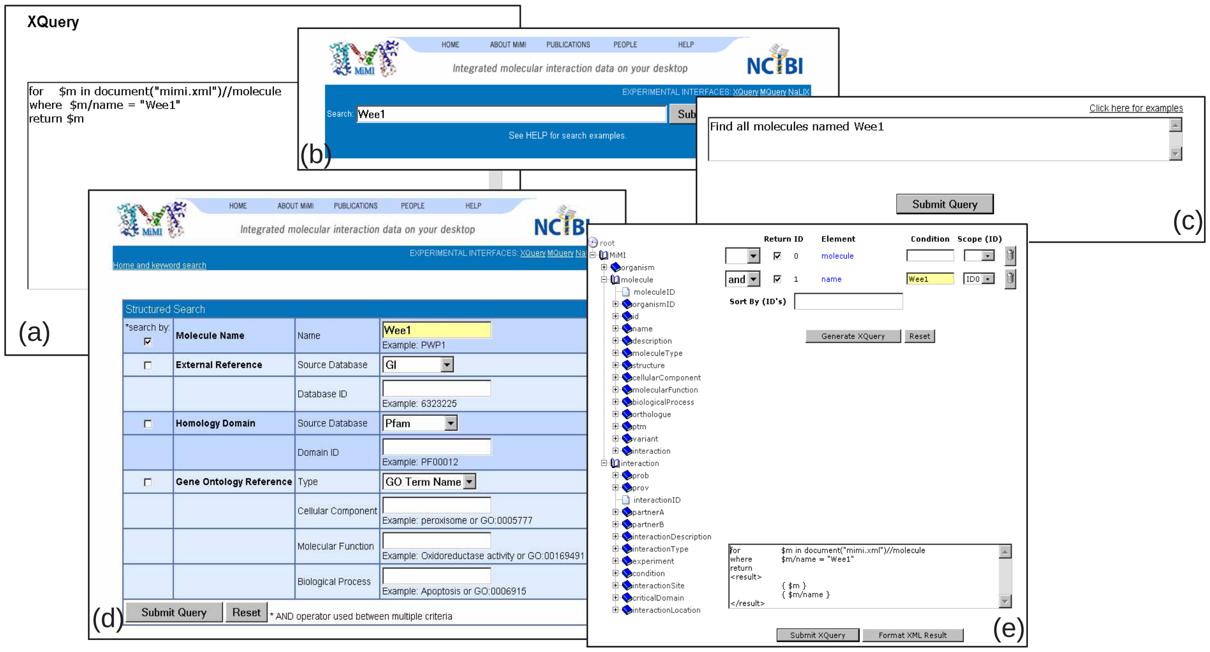
Given the XML representation of MiMI data, XQuery was our first choice for accessing the database. Indeed, some users wanted the power of a declarative query language, even if they didn't have the training to write such queries. A majority of users, however, were complete technophobes and preferred forms-based interfaces. (Such interfaces do a good job today for specific applications—quite complex back-end queries can be run, for instance in an airline reservations database, while the user is shielded from this complexity by a simple form-based query interface.) Aside from these were a few users who wanted to download the entire dataset and write Perl scripts to slice and dice it. Our challenge in MiMI was to provide easy-to-use interfaces beyond a few hand-designed forms for some common queries. In fact, MiMI allows users to access data through various interfaces, which are depicted in Figure 1 and discussed in greater detail in the next section.

3.1 Accomplishments

We started with query interfaces at two opposite ends of the spectrum: XQuery and a simple forms-based interface. Almost right away, we decided to add a visual query builder, MQuery, as an intermediate option between the two opposites. MQuery enables users to create declarative queries incrementally by clicking on elements of interest in a graphical schema tree and filling in form fields associated with each of them. However, we found that only a few people preferred this interface—it was considered not much simpler than writing XQuery in many circumstances.

It turned out that users' difficulties with both XQuery and MQuery are caused not only by the syntax of the language, but more importantly by the mere complexity of the MiMI schema. Almost all users found it difficult to locate elements of interest to be used in their query specification. To assist such users we aggressively tackled this problem with multiple approaches. One approach we developed is that of schema summarization [97]. The idea is to develop a representation of the underlying complex schema at different levels of detail. A user unfamiliar with the database would first be shown a high-level schema summary comprising only a small number of concepts. Progressively greater detail is revealed on demand as the user zooms in on the portions of the schema of greatest interest. Based on the schema summary concept, we have recently developed a new query model called Meaningful Summary Query [98], which allows a user to query the database through the schema summary directly, without the knowledge of the underlying complex schema and with high result quality and query performance.

Another approach we developed was that of Schema-Free XQuery [62, 63]. The idea here is to allow users to specify query entities of interest without specifying how they are



XQuery
(a)
HOME ABOUT MIMI PUBLICATIONS PEOPLE HELP
MiMI
NCIBI
Integrated molecular interaction data on your desktop
EXPERIMENTAL INTERFACES: XQuery MQuery NaLIX
(b)
Click here for examples
Submit Query
(c)
Home and keyword search
(d)
(e)

```
for    $m in document("mimi.xml")//molecule
where  $m/name = "Wee1"
return $m
```

Search: Wee1

See HELP for search examples.

Find all molecules named Wee1

Structured Search

*search by: ✓	**Molecule Name**	Name	Wee1 Example: PWP1
☐	**External Reference**	Source Database	GI
		Database ID	Example: 6323225
☐	**Homology Domain**	Source Database	Pfam
		Domain ID	Example: PF00012
☐	**Gene Ontology Reference**	Type	GO Term Name
		Cellular Component	Example: peroxisome or GO:0005777
		Molecular Function	Example: Oxidoreductase activity or GO:00169491
		Biological Process	Example: Apoptosis or GO:0006915

Submit Query Reset * AND operator used between multiple criteria

```
root
MiMI
  organism
  molecule
    moleculeID
    organismID
    id
    name
    description
    moleculeType
    structure
    cellularComponent
    molecularFunction
    biologicalProcess
    orthologue
    ptm
    variant
    interaction
  interaction
    prob
    prov
    interactionID
    partnerA
    partnerB
    interactionDescription
    interactionType
    experiment
    condition
    interactionSite
    criticalDomain
    interactionLocation
```

	Return ID		Element	Condition	Scope (ID)
	✓	0	molecule		
and	✓	1	name	Wee1	ID0

Sort By (ID's)

Generate XQuery Reset

```
for          $m in document("mimi.xml")//molecule
where        $m/name = "Wee1"
return
<result>
             { $m }
             { $m/name }
</result>
```

Submit XQuery Format XML Result

Figure 1: Methods users can interact with MiMI: (a) XQuery (b) Keyword (c) Natural Language (d) Forms (e) MQuery.

hierarchically related. The system then automatically determines local structures in the XML database that match these specifications. The users can specify queries based on whatever partial knowledge of the schema they have—knowing the full schema, they can write regular XQuery statements; not knowing the schema at all, they can just specify labeled keywords; most importantly, they can specify queries somewhere in between. The system will respect whatever specifications are given. More recently, we have extended this work to Ranked Relatedness Queries [35], which return a set of matches with associated relevance scores.

For certain types of queries, our users, being accustomed to web search, preferred keyword-based search. We promptly added such an interface to MiMI. All textual fields within the database were used to populate a Lucene index [44]. One challenge we faced was that there were multiple alternative spellings of the names of an entity (i.e., a gene or a protein) and names were quite long. Frequently, a user issued a keyword query only to find later that the results were incorrect because the wrong keywords were used. An instantaneous feedback mechanism was highly desired. This desire led us to the creation of the word autocompletion interface, which had since evolved into a phrase autocompletion interface [75], and more recently, into a query autocompletion facility. In particular, the query autocompletion facility is being demonstrated at this conference [74].

Finally, some of our users, being complete technophobes, preferred to access MiMI in their own language, English. We developed an interactive Natural Language Interface for Querying XML, NaLIX [58, 59], which is built on top of Schema-Free XQuery. NaLIX is a generic natural language query interface capable of handling queries with not just selection, but also joins, nesting, and aggregation. It uses a directed iterative refinement method to assist users in restating queries when it is unable to understand the users' natural language statements [60]. We extended NaLIX to permit conversational querying so that users can construct complex queries as modifications of previously issued queries [61]. We also added a domain learning component to the generic NaLIX system, leveraging off the iterative restatement of queries. This system, DaNaLIX is being demonstrated at this conference [57].

In short, we developed a rich panoply of interfaces with which to access MiMI. Our intention was that each user could choose to interact with the database using the interface they prefer. Many issues, however, still remain.

3.2 Remaining Issues

Many of our users have strong preferences for some interfaces over others. But users do access MiMI through multiple interfaces. Somewhat surprisingly, we receive complaints about inconsistencies between different interfaces—some users found different results by going through different interfaces, as the following example shows.

Example 1. *A user issues a keyword query 'Wee1' through the keyword interface. The query evaluation accesses the Lucene index, which is constructed on all textual fields, irrespective of which field it occurs in. Hundreds of results are returned, including molecules that bear some relationship to Wee1 and mention the string 'Wee1' somewhere in one of their fields. Later on, the same user issue a form-based query through the MQuery interface, by typing 'Wee1' into the 'Molecule Name' field, and only ten results are returned—the molecules that contain Wee1 as their name. The user complains to us for producing inconsistent results.*

While we, as computer scientists, can see immediately where the problem is, our users don't. And the burden is on us to explain to our users, in an effective way, the reasons behind this inconsistency.

Another set of complaints we frequently get are the inability to explore and manipulate the data directly, in a graphical setting. We partially addressed this issue by integrating a popular graphical tool, Cytoscape [86], into MiMI. While users cannot issue complex queries over the database because of the limitations of Cytoscape, they are happy to be able to graphically manipulate the results, which are viewed as a graph of interacting protein nodes. One primary benefit of using Cytoscape is the ease of specifying joins to find related interactions or proteins in the graphical setting. (In fact, we found joins to be extremely hard for our users to reason with correctly.) How to allow complex query semantics over a graphical representation of the data is a major research issue that we are currently pursuing.

Finally, many of our users frequently generate scientific results from the experiments performed in their labs—results that they would like to put into MiMI for easy access by others. However, MiMI's rigid structure, as exemplified by a schema that is updated only a few times a year, prevents users from simply putting their data into MiMI. Instead, they need to understand the MiMI structure first and convert their data to the MiMI format before the update can be made. In reality, little data has been uploaded into MiMI from our users primarily because they are all busy scientists whose time is simply too precious to be spent understanding the schema of a database they use as a tool.

As we analyze our accomplishments, and more importantly, the many remaining issues described above, we have come to realize that the usability of a database system is much more than skin deep. Our work on query interfaces may contribute towards the usability of a system, but they are far from enough to provide the optimal user experience. In the next section, we enunciate what we believe are the major database usability problems.

4. THE PERSISTENCE OF PAIN

When we look at how users struggle with database systems today, we see several major issues. Whereas we have certainly been motivated to study issues of usability because of MiMI and our biological collaborations, we believe that the usability concepts we present in this paper are applicable universally and not just to scientific data. To stress this point, and to make this paper accessible to the database researcher who may not know much Biology, we have chosen an airline database as our running example in this paper and avoided the use of biological terms.

4.1 Painful Relations

Whereas a single table of data is natural for most people, joins between multiple tables are not. Unfortunately, normalization is at the center of relational design. Indeed normalization saves space, avoids update anomalies, and is a desirable property from many perspectives. However, the use of joins in a relational model does not retain the integrity of data objects that a user regards as one unit.

Consider an airline database with a basic schema shown in Figure 2, 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 2, there is the possibility of trouble. Obviously, the *airports* table has information about the starting location and the destination. However, how do we figure out what is used by a particular flight? The words fid and tid have no meaning. Instead we must bring up the schema, press our fingers to the monitors, and create a greasy smear as we follow the foreign key constraint. Alternatively, we could suffer almost as much pain by reading the database creation statements for the same information. The current solution to manage this pain is to hire DBAs and offer them copious amounts of money not to leave once they have learned the company's database schema well.

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 updatable. However, the users will have to stitch the information back together to answer most of the real queries. The fundamental issue is that joins destroy the connections between information pertaining to the same real world entities and are nonintuitive to most normal users. We note that many commercial database systems carefully denormalize their schema to reduce the number of joins required, although the purpose there is to speed up query evaluation.

Finally, queries become painful to express across multiple tables. Because joins innately disrupt data cohesion, such queries are problematic for many users. For example, consider a query as simple as "Find all flights from Detroit to Beijing" in our airline database. Even though we are interested only in information about *flights*, the city names that specify the selection predicate are found only in the *airports* relation, which must be joined twice with $flight\_info$ to express our query.

For queries like this one with only a few simple joins, it is not so hard to see how one could reconstruct the underlying object that was broken apart to produce the normalized schema. One could even envisage an automated tool to do so. However, when we start having recursive self-joins or one-to-many joins with variable cardinality, or non-equijoins, it is not even clear how to produce a single tuple for the user to operate on in a tabular fashion. In fact, even experts may run into trouble, as pointed out by David Beech [6], the technical guru at Oracle: "Supposing that the same set of features is widely available in different implementations, will the standards be well enough understood by users who are not programming wizards? I'm not implying that application developers will all use SQL directly. Even if higher-level tools conceal the syntax of the language, users must clearly understand the data model or type system of the manipulated information."

Some practitioners have realized that typical commercial databases are too heavy duty and too confusing for the average user. Approaches such as DabbleDB [31] or OpenRecord [76] present users with "easy" relational systems. However, even these systems, specifically geared toward taking the onus of SQL away from the user, are still haunted by the

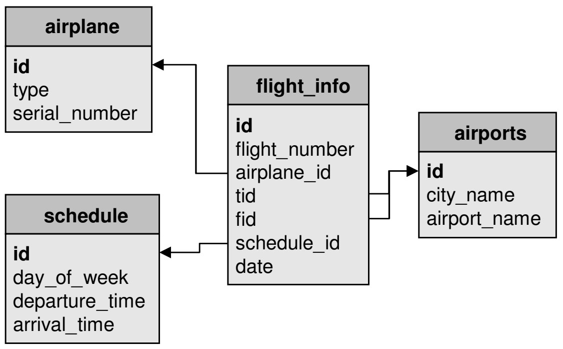


Figure 2: The base tables needed to store a "flight". A flight contains from location, destination, airplane info and schedule, yet consists of at least four tables. Note that an actual schema for such data is likely to involve many more attributes and tables.

join curse. The normalization/join notion creeps into even the best interfaces because of its centrality in the relational mind set.

4.2 Painful Options

Most computer scientists will consider a system that supports both A and B as being superior to a system that only supports A, for any functionality A and B. After all, the first system can emulate the second, while the second cannot emulate the first. Similarly, a system that permits adjustment of some tuning parameter is logically superior to another that does not permit this parameter to be adjusted.

Unfortunately, this sort of logic quickly leads us to create software with too much functionality and too many options. The problem with irrelevant options is that they compete with the relevant ones. Exercising options is not costless. A cost analysis of weighing diverse options was performed in [88]. Psychologists can also tell us about regret regarding "paths not taken," a cost that would not be there if the alternative paths were not there in the first place. [48] studies the effects of forgone options and models its cost to the user. Too many choices can also have an adverse effect on a user's "need for closure" which is studied in [67]. A great deal of the mystique of Apple Inc. has to do with their reducing the functionality of their product, and hence making them better. Witness the success of the iPod: the simplest set of features that can possibly work is what attracts large numbers of users—not the latest in whizbang gadgetry [81].

The database metamodel today is all about options. Query optimization is at the heart of what we do. Determining algebraic equivalences and generating alternative query plans is at the core of our heart. The same query can be posed multiple equivalent ways in SQL.

Instead, we should design systems for customized value and care only about how well users can get their job done. Unexplored alternatives provide no value, as the nervous interviewee was told during job search, "It is not the number of offers that counts, but rather the quality of the one offer that you accept." It is no surprise that forms-based interfaces, for all their limitations, have been the primary means through which non-experts interact with databases.

Forms limit database access to a specific set of queries felt to be of most use to their target users. They offer a convenient entry-point to the database that requires no knowledge of query languages or data organization that other querying mechanisms like SQL require to obtain the same information. But this simplicity comes at a price. While we would like to limit options, it is not easy to determine which options to keep and which options to leave out. "What do users want?" is a difficult question to answer, especially since users are not all the same. While cutting down on options can greatly enhance the querying experience for some users, it might dissatisfy many others. The challenge is to simplify querying for novice users, while providing the expert user with the tools she needs to be productive.

4.3 Unexpected Pain

Another place where database systems can frustrate users is when they produce results that are unexpected with no explanation. Shielded from the details of the system, a user can obtain results that do not make sense to her because her mental model and assumptions of the system conflict with the actual underpinnings of the system. A counter-example is a web search engine, which has a complex ranking procedure for search results and does not reveal the technical details to most users. Users seem to accept search engines because: 1) Expectations are set correctly. Search engines are upfront about performing behind the scenes magic that the user cannot influence. 2) Usually, the top few results returned do contain what the user is looking for. 3) If the result is not there, the user has the option to either trawl through pages of returned options, or to try a different query path. 4) The web is huge, and no one knows exactly what is out there. If a user is not presented with a relevant page, chances are he would not know he missed it, and would make do with a different entry.

Unfortunately, the search engine strategy will not work for database systems mainly because of (4). When a user queries an airline reservation system, she may know that there are flights between Detroit and Beijing. When the system tells her otherwise, it is unexpected and she will demand an explanation. This unexpectedness comes in two forms, *unable to query* and *unexpected results*. We have mentioned the latter previously. We now describe both of them in more detail.

4.3.1 Unable to Query

When a system hinders users from querying the data in the way that they want, it frustrates the user. It is especially frustrating when the user knows that the underlying data exists, yet she cannot query it:

EXAMPLE 2. *Consider a world traveler who has infinite flexibility, many destinations to visit but limited money. She visits her favorite airline reservation site, and chooses "Flexible Dates". After filling in a bit more information, she attempts to specify multiple stops. Suddenly, she is forced to enter fixed dates into the system!*

The system behaved unexpectedly, which led to the user not being able to construct her query, but why? Was it because she cannot have multiple hops with flexible dates? Or was it some other piece of information she entered? Even if the user determined that she could not enter flexible dates with multiple hops, there was still unexpectedness. The data obviously exists in the database, so has she merely chosen the wrong interface to access it? If she had started in a different place, or followed a different path, could she have fulfilled her request?

This is a fundamental problem with forms-based interfaces. Forms, by definition, provide only a limited view of the underlying data. When users' mental model of the data differs from that of the form designer, unexpected pain results. The user has to focus on *how* to obtain results rather than on *what* results they want. This practically eliminates the benefit of declarative querying that an RDBMS provides.

The need for application-specific forms-based interfaces stems from the general database usability problems. If the average user could successfully query a relational database, there would be no need for a separate form for each application. Ideally, the system could adapt to the user's view of what the data represents and allow the user to query it in whatever way makes sense for her.

Developing methods for querying relational databases that are agnostic with respect to the true underlying structure poses many challenges. First, the system must determine what the user wants to query for. In the cases of simple selection conditions, this determination is not too hard so long as the user knows what fields are available and the system knows how to join the tables containing these fields. In the case of queries requiring aggregation and multiple complex joins, it is not so clear how to provide the user a straightforward yet comprehensive way of specifying what they want to query.

Even if the user successfully specifies what to query, the system may be unable or unwilling to perform the query. The reasons for this are manifold: performance or security concerns, data or program errors at some level of the system, etc. To avoid unnecessary pain, the system must be able to report these failures to the user in a meaningful way. Determining the appropriate level of detail so as to help the user without overwhelming her is a challenge in and of itself.

4.3.2 Unexpected Results

There is still plenty of opportunity for unexpected pain even when a user is able to successfully navigate through a system's query interface. When the user encounters unexpected results and no explanation is provided, the user is again frustrated. Previous work [70] focuses on explanation for empty results. However, even non-empty results can still be unexpected. We present two different ways that users can encounter problems with the results. The first is with the base data itself.

Example 3. *A user books a trip through the airline reservation system and requests lowest fare and a window seat. However, the system keeps giving him an aisle seat without any error message. Where does the aisle seat come from? Is it from the pool of general seats or is it from the pool of seats with the lowest fare?*

Often the need to know where a result comes from is only requested when something goes wrong. However, it can be necessary in its own right. For instance, you may be interested in the list of prohibited items published by the government agency TSA, but not by the individual airlines. It matters *where* a result comes from. However, *where* alone is not enough, as the next example will demonstrate.

Example 4. *A user, looking for an escape, peruses the list of cheap flights provided by her favorite airline. She can get to Los Angeles for $75, Boston for $100 and San Francisco for $400. Why is San Francisco on this list? It is not a particularly cheap fare, but it must have satisfied some criteria to be placed there.*

In addition to where a result comes from, *why* a result is returned is also essential. The latter describes why a particular item is included in a set [8, 30]. For instance, in the above example, if the criteria for inclusion is that the fare is less than the average flight price for the next month, and the San Francisco fare satisfies this, it should be included on the list.

When users encounter unexpected results, it is responsibility of the database system to explain to them the *where* and *why*. The usability of the system can be significantly affected when no such explanation can be given.

4.4 Unseen Pain

As computer scientists, when we think of database users, our instinct is that they will think like us. But we are not typical database users. For example, the vast majority of us today prefer to use LaTeX for document creation. Yet an overwhelming majority of the rest of the world prefers Microsoft Word. As a computer scientist, you can explain why you prefer LaTeX to Word—the former is elegant, it permits global changes more easily, it separates content from formatting, it stores everything in small ASCII files, and so on. Yet, Word has one overwhelming advantage over LaTeX—it has the "What You See is What You Get" (WYSIWYG) property. As a LaTeX user, you edit the source file and *predict* what your modifications will do to the output generated. If you are an experienced LaTeX user, and have your brain wired like a Computer Scientist, your predictions are correct most of the time. If your predictions are correct often enough, your few mistakes are easy to take in stride. For the lay user, though, this can become a frustrating barrier to use.

Our situation with database manipulation is similar. Essentially all query languages, including visual query builders, separate query specification from output. A user issues a query, and hopes that it will produce the desired output. If it does not, then she has to revise the query and resubmit. There has been some discussion in the database community of query sequences, but the assumption is that a query is being reformulated because the user is "exploring" the data. While this may be true in some cases, often the query reformulation is because the user did not initially specify the query correctly.

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 Inchon airport to be suggested no matter whether the user entered Inchon or Seoul or even Soul. This does not seem too hard, and some airline web sites will do this. But now consider a user who wants to visit KAIST, and so enters Daejeon as the city to fly to. No search interface today, to our knowledge, can suggest flying into Inchon airport even though that is likely to be the preferred solution for most travelers.

A significant part of database query specification is result *construction*. Once the FROM and WHERE clauses of a SQL query have been executed, we have data in hand that must be manipulated to produce the desired output. In the case of report generation from a data warehouse, there may not even be a selection condition to apply—the entire query specification is about how to aggregate and present the results. Indeed, the only examples we are aware of that provide WYSIWYG capabilities in the database context are warehouse report generation tools.

What does WYSIWYG mean for databases? After all, the point of specifying a query is to get at 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 [74] 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.

Other examples of WYSIWYG in databases can be seen in a geographical context. 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 travel sites already provide a facility to specify dates using a pop-up calendar. It is just a question of taking this WYSIWYG approach and pushing it farther. 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.

4.5 Birthing Pain

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. It is not because users do not want to store their information inside a database. Rather, there are many everyday data a user would like to put into her databases [7] such as shopping lists, expense reports, etc. The main reason for this "birthing pain" is that creating a database and putting information into a database are not easy tasks. For example, creating a database in current systems requires a careful design of the database schema, for which ordinary users simply do not have the inclination or expertise. Similarly, putting information into an existing database may require the user to re-organize her information according to the specific structure in this existing database, which involves understanding this structure and developing a mapping to it from the data.

EXAMPLE 5. *Consider our 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.

5. THE PAINLESS FUTURE

When we speak of usability, we mean much more than just the user interface, which is only a part of the usability equation. A more fundamental concern is that of the underlying architecture. To understand this, consider computer system security as a parallel example. When we think of security, many would immediately think of firewalls, and indeed firewalls are an important part of establishing a secure computing environment. Yet, we all appreciate that a firewall in itself is not enough—security is truly obtained only when it is designed into every aspect of the system.

While they provide visual means to let users manipulate queries easily, state-of-the-art query builders and graphical interfaces on top of current database systems still require abstraction of the query semantics through the user—something at which she may not be particularly adept. They also tend to expose the underlying database schema to the user, adding to her cognitive burden. Furthermore, there are few friendly ways for a user to create or edit a database.

We need database systems that reflect the user's model of the data, rather than forcing the data to fit a particular model. Even if we have a relational implementation under the hood, it should be hidden from the user, who should see the data presented in a form that is "natural." This means there is no single standard presentation data model of data. However, there are at least a few major models that work well to model significant segments of user applications:

- *Geographic:* Many data sources of interest have geographic or spatial distributions. These include not just traditional geospatial data and map data, but also any information with a location component. In fact, on the web, mashups have been tremendously successful in presenting joins between data sets using a geographic location as the basis.
- *Network:* We may have a graph or network representation of data that is natural in many circumstances. In the case of MiMI, we found that for many scientists, protein interaction data is most naturally viewed as a graph. This is the case even when the scientist is not directly interested in graph properties: for example, when viewing the properties of a single interaction between a pair of proteins, scientists still prefer to view the interaction as an edge in the graph. We conjecture that this preference is because the local neighborhood of the graph establishes "context" for the scientist and provides her with confidence that she is indeed looking at the correct interaction. Moreover, it is also much easier to point and click than to type.
- *Multidimensional:* The multidimensional data model is the presentation data model that is perhaps the most successful commercially. It was explicitly called a data model, and introduced for decision support queries on warehoused data almost fifteen years ago [82]. It has since been adopted widely, and even today it is at the cutting edge of database user interaction [43, 19]. While data in the warehouse itself may be stored in a star schema with multiple tables, users of the multi-dimensional data model think of the data as points in multi-dimensional space, with aggregates of measure attributes being computed over specified ranges of dimension axes.
- *Tabular:* While joins across multiple normalized tables may be difficult, people are certainly used to seeing data represented in simple two-dimensional tables. The popularity of the Excel spreadsheet as a data model speaks to this. For situations where data can be represented conveniently as a table, a tabular model is certainly appropriate.

The concept of a view has been around almost since the beginning of relational data management. Traditionally, this has been just another relation, defined as the result of evaluating a query. Given the need to support various **presentation data models**, including those just listed, we can generalize the notion of a view to be not just a table, but a representation of derived information in the presentation data model. Manipulating data through the presentation data model leads to the well-known problem of updating through views [39]. There are excellent research problems to be addressed in characterizing presentation data models and types of updates that can be supported without ambiguous updates.

Given a data set, it may not always be the case that a single presentation data model is best to serve all user needs. For example, most travel sites will show hotel options in both a geographical view and a textual list view. Each view has its strengths, and most users seem to have no trouble handling this choice of views. However, there is an expectation of **consistency among view options**—if a selection is applied, say on price, in the list view, the user should expect that selection to be reflected in the geographic view. If these are implemented as views on the underlying data in the manner we suggested in the preceding paragraph, then this type of consistency should be maintained automatically. Furthermore, it should be noted that while having two options for views may be appreciated by users, it is probably the case that eight options would be considered too much.

In addition to consistency, notions of **data provenance** must be integrated into the presentation data models. Provenance [10, 18, 96], both *why* provenance and *where* provenance, can assist the user in understanding the results presented to her. Most discussions of provenance today are in the context of scientific workflows and scientific data management [38, 78, 95]. However, data provenance is important for most application domains, including everyday tasks such as travel planning and weather monitoring. Provenance usability is still in its infancy and presents fertile ground to explore. To successfully include provenance in any system, we must find an easy, automated and unobtrusive way to capture it. Recent work [17, 72] present initial attempts at this. However, how to succeed in capturing the correct information, unobtrusively throughout the entire system is still an open topic. Moreover, once captured, the amount of provenance can easily outweigh the size of the data itself. Good provenance storage, compression, and query mechanisms need to be in place. Finally, there must be a way to make this provenance information understandable to the user: [25, 27] present provenance viewing strategies, but understandability is far more than just easily viewing the provenance entries.

To allow more intuitive user interaction with the database, the presentation data model should be capable of **direct data manipulation**. Users are very good at point-and-click, drag-and-drop, and, to a lesser extent, filling in textboxes. But, whatever they do, they should not be surprised by what they get. In addition, we must develop an algebra of operations in the presentation data model such that the basic needs of most users are met by a very small number of operators, thus reducing the barrier to adoption. As users gain experience with the data model, they can become more proficient at manipulating it, and can add to the suite of operators they know how to invoke, thereby increasing the expressive power of the algebra.

Finally, database systems must accommodate users who expect to create and update their databases and yet have no interest or expertise in database design and database integration. In spirit similar to the Dataspace concept [42, 65], we argue that database systems need to support interfaces for casual **"schema-later" and "heterogeneous" database design**: a database can be created with data that is unstructured or (heterogeneously) structured, and the system needs to take advantage of whatever structure the data currently has. Furthermore, the system needs to provide functionality for users to add structure easily when there is a need and in a manner convenient to them.

6. CONCLUSION

Database systems today, for all their virtues, are extremely difficult for most people to interact with. This difficulty cannot be fixed just by improving the query interface. Rather, we must rethink the architecture of the database system as a whole. This paper has suggested a framework for this purpose comprising a presentation data model as a distinct layer above the usual logical data model. We envision this presentation data model to allow effective user interaction with the database through direct data manipulation.

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

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

[6] D. Beech. Can SQL3 Be Simplified? *Database Programming and Design*, 10(1):46–50, Jan 1997.

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

[8] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *VLDB*, 2006.

[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] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. In *VLDB*, 2005.

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

[12] A. Blum. Microsoft English Query 7.5: Automatic Extraction of Semantics from Relational Databases and OLAP Cubes. In *VLDB*, 1999.

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

[14] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7):107–117, 1998.

[15] A. Brown, L. Chung, W. Kakes, C. Ling, and D. Patterson. Experience With Evaluating Human-Assisted Recovery Processes. *Dependable Systems and Networks*, pages 405–410, 2004.

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

[17] P. Buneman, A. Chapman, and J. Cheney. Provenance Management in Curated Databases. In *SIGMOD*, 2006.

[18] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT*, 2001.

[19] Business Objects, Inc. Crystal Xcelsius, http://xcelsius.com.

[20] 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.

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

[22] 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.

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

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

[25] K. Cheung and J. Hunter. Provenance Explorer - Customized Provenance Views Using Semantic Inferencing. In *ISWC*, 2006.

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

[27] S. Cohen, S. C. Boulakia, and S. Davidson. Towards a Model of Scientific Workflows and User Views. In *DILS*, 2006.

[28] 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.

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

[30] Y. Cui and J. Widom. Lineage Tracing for General Data Warehouse Transformations. In *VLDB*, 2001.

[31] DabbleDB. http://www.dabbledb.com/.

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

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

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

[35] A. Elkiss, Y. Li, and H. V. Jagadish. Ranked Relatedness Queries for XML Databases. Technical report, University of Michigan, 2007.

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

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

[38] J. Frew and R. Bose. Earth System Science Workbench: A Data Management Infrastructure for Earth Science Products. In *SSDBM*, 2001.

[39] A. Furtado and M. Casanova. Updating relational views. In *Query Processing in Database Systems*, 1985.

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

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

[42] A. Y. Halevy, M. J. Franklin, and D. Maier. Principles of Dataspace Systems. In *PODS*, 2006.

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

[44] E. Hatcher and O. Gospodnetic. *Lucene in Action.* Manning Publications, 2004.

[45] 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.

[46] T. Haveliwala, S. Kamvar, and G. Jeh. An Analytical Comparison of Approaches to Personalizing PageRank, Preprint, June 2003.

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

[48] J. J. Inman, J. S. Dyer, and J. Jia. A Generalized Utility Model of Disappointement and Regret Effects on Post-Choice Valuation. *Marketing Science*, 16(2):97–111, 1997.

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

[50] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *VLDB Journal*, 11(4):274–291, 2002.

[51] M. Jayapandian, A. Chapman, V. G. Tarcea, C. Yu, A. Elkiss, A. Ianni, B. Liu, A. Nandi, C. Santos, P. Andrews, B. Athey, D. States, and H. Jagadish. Michigan Molecular Interactions (MiMI): Putting the Jigsaw Puzzle Together. *Nucleic Acids Research*, pages D566–D571, Jan 2007.

[52] 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.

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

[54] Y. Kanza and Y. Sagiv. Flexible Queries Over Semistructured Data. In *PODS*, 2001.

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

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

[57] Y. Li, I. Chaudhuri, H. Yang, S. Singh, and H. V. Jagadish. DaNaLIX: a Domain-adaptive Natural Language Interface for Querying XML. In *SIGMOD*, 2007.

[58] Y. Li, H. Yang, and H. V. Jagadish. NaLIX: an Interactive Natural Language Interface for Querying XML. In *SIGMOD*, 2005.

[59] Y. Li, H. Yang, and H. V. Jagadish. Constructing a Generic Natural Language Interface for an XML Database. In *EDBT*, 2006.

[60] Y. Li, H. Yang, and H. V. Jagadish. Term Disambiguation in Natural Language Query for XML. In *FQAS*, 2006.

[61] Y. Li, H. Yang, and H. V. Jagadish. NaLIX: A Generic Natural Language Search Environment for XML Data. *acmtds*, accepted.

[62] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.

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

[64] 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.

[65] 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.

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

[67] I. Mervielde. The Need for Closure and the Spontaneous Use of Complex and Simple Cognitive Structures. *The Journal of Social Psychology*, 2003.

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

[69] B. Mobasher, R. Cooley, and J. Srivastava. Automatic Personalization Based on Web Usage Mining. *CACM*, 43(8):142–151, 2000.

[70] A. Motro. Query generalization: A method for interpreting null answers. In *Workshop on Expert Database Systems*, 1986.

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

[72] S. Munroe, S. Miles, L. Moreau, and J. Vázquez-Salceda. PrIMe: A Software Engineering Methodology for Developing Provenance-Aware Applications. In *SEM*, 2006.

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

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

[75] A. Nandi and H. V. Jagadish. Effective Phrase Prediction. Technical report, University of Michigan, 2007.

[76] OpenRecord. http://www.openrecord.org/.

[77] D. Oppenheimer. The Importance of Understanding Distributed System Configuration. *System Administrators are Users, Too: Designing Workspaces for Managing Internet-scale Systems, CHI 2003 Workshop*, 2003.

[78] C. Pancerella, J. Hewson, W. Koegler, et al. Metadata in the Collaboratory for Multi-Scale Chemical Science. In *Dublin Core Conference*, 2003.

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

[80] M. Perkowitz and O. Etzioni. Adaptive Web Sites. *CACM*, 43(8):152–158, 2000.

[81] A. Pfeiffer. Why Features Don't Matter Anymore: The New Laws of Digital Technology. *Ubiquity*, 7(7), Feburary 2006.

[82] Pilot Software. http://www.pilotsoftware.com/.

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

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

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

[86] P. Shannon et al. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Res*, 13(11):2498–504, 2003.

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

[88] S. M. Shugan. The Cost of Thinking. *Journal of Consumer Research*, 7(2):99–111, 1980.

[89] Y. Simmhan, B. Plale, and D. Gannon. A Survey of Data Provenance in E-Science. *SIGMOD Record*, 34(3):31–36, 2005.

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

[91] 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.

[92] 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.

[93] A. Tornqvist, C. Nelson, and M. Johnsson. XML and Objects-The Future for E-Forms on the Web. In *WETICE*. IEEE Computer Society, 1999.

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

[95] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *CIDR*, 2005.

[96] A. Woodruff and M. Stonebraker. Supporting Fine-grained Data Lineage in a Database Visualization Environment. In *ICDE*, 1997.

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

[98] C. Yu and H. V. Jagadish. Querying Complex Structured Databases. Technical report, University of Michigan, 2007.

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

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