A Relationally Complete Visual Query Language for Heterogeneous Data Sources and Pervasive Querying

Stavros Polyviou
George Samaras
Paraskevas Evripidou
Department of Computing
University of Cyprus
75 Kallipoleos St.
1075 Nicosia, Cyprus
Email: {polyviou, skevos, cssamara}@ucy.ac.cy

Abstract— **In this paper we introduce and formally define Query by Browsing (QBB), a scalable, relationally complete visual query language based on the desktop user interface paradigm and tuple relational calculus that allows the formulation of complex queries over relational, entity-relationship, object-oriented and XML data sources on a variety of handheld and desktop platforms. It is to our knowledge the first visual query language to combine the important characteristics of usability, scalability, expressive power and flexibility. We support these claims by demonstrating the similarity of the QBB paradigm to the popular desktop user interface paradigm, by relating it to relational calculus and relational algebra and by describing Chiromancer II, a web-based implementation of the QBB paradigm for handheld devices. We also discuss ways in which non-relational sources can be represented and queried and compare QBB to related work in the area of visual query languages for a variety of data models. We finally offer conclusions and thoughts for future work.**

I. Introduction

As the volume and heterogeneity of data increases, as users become more demanding in their queries and mobile in their workplace and as information is accessed through an increasing variety of devices, the need for better search interfaces becomes imperative. We believe that a modern search interface able to deal with these challenges should be:

- Usable: discretionary users should be able to form meaningful queries with little or no instruction and within a reasonable amount of time.
- Powerful: queries of arbitrary complexity should be achievable.
- Flexible: suitable for a wide variety of domains and data models.
- Scalable: able to accommodate large schemas with large amounts of information and adaptable to the diminutive screen size of handheld devices.

In this paper we present Query by Browsing (QBB), a Visual Query Language that combines the file browsing features of the desktop user interface paradigm with concepts from the relational model and the expressive power of relational calculus. We begin by describing and formally defining the QBB paradigm. We then demonstrate its expressive power by comparing it to relational algebra and make claims about its usability by presenting its similarity to the widely popular desktop user interface paradigm. We discuss its suitability for presenting and querying non-relational data sources such as entity-relationship, object-oriented and XML schemata. We also compare and contrast it with other query languages in terms of ease of use, expressiveness, flexibility and scalability. We then provide some implementation details regarding Chiromancer II and the QBB paradigm in general. Throughout our discussion, we use screenshots from Chiromancer II, a web-based implementation of the QBB paradigm for handheld devices, thus demonstrating its scalability through its minimal screen requirements. Finally, we present our conclusions and thoughts for future work.

II. The QBB paradigm

The main idea behind the QBB paradigm is to present the user with an interface that looks very similar to that of a file browser based on the desktop user interface paradigm. Users encounter the same kinds of objects they normally see in a modern file browser such as folders, documents and shortcuts, and are able to perform the same actions on those objects such as creating or deleting them, opening and closing them and copying or moving them. In QBB however, these objects and actions have the semantics of relational model objects (relations, attributes, tuples) and operators (projection, restriction, join etc). The idea is to add these semantics in such a way so that the QBB paradigm appears as an extension of the desktop user interface paradigm rather as an alternative to it. This way we can capitalize on user's familiarity with that paradigm, while harnessing the expressive power and elegance of the relational model.

The other important characteristic of QBB is the strong emphasis placed on schema browsing. The ability to present

Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)
1084-4627/05 $20.00 © 2005 **IEEE**

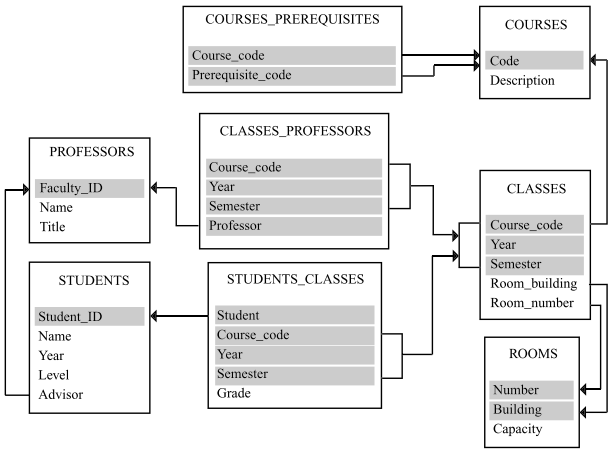

Fig. 1. Diagrammatic representation of the sample database schema. Primary key columns are highlighted. Arrows start from foreign key columns and point to the corresponding primary key columns.

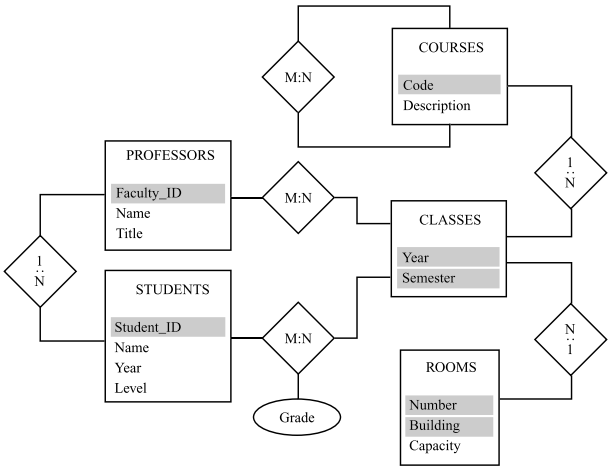

Fig. 2. Entity-Relationship diagram representing the relational schema in figure 1.

the database schema to users who are unfamiliar with the database is one of the strong points of visual query languages [1]. QBB takes the concept to a new level, not only by placing it at the center of the user experience, but also by allowing the construction of multiple and flexible views on the same schema. To that end, some additions to the desktop user interface paradigm had to be made. We believe that these additions offer significant advantages in the areas of usability and scalability that outweigh any disadvantages due to the learning curve they introduce to existing desktop paradigm users.

Last but not least, QBB can function as a framework, and as such can accommodate custom solutions for specialized domains such as geographical information systems, multimedia libraries and so forth. It adopts many ideas found in the area of dynamic queries [2] and has been designed in such a way so as to accommodate many of the interfaces and architectures proposed in that area.

A. An illustrative example

Let us introduce the QBB paradigm by presenting a simple example. Consider the schema of a relational database describing the relationships between students, professors, courses, classes and rooms, shown in figure 1. We can easily translate this relational schema into an equivalent Entity-Relationship diagram, as shown in figure 2. We proceed then by representing every entity (or relationship that has at least one associated attribute) as a folder and each relationship as a parent folder/subfolder relationship. We thus create a forest of trees, each tree representing all possible paths through the Entity-Relationship diagram starting from each one of the entities in the diagram. Since our diagram contains cycles, parts of these trees will end up having an infinite depth. A partially expanded forest for the schema in figure 1 is shown in figure 3.

Since a folder represents an entity (or relation), it follows then that it must contain a set of attributes and a set of records representing the instances of that entity. A root folder contains all rows from the table it represents. However, a subfolder representing the same table is restricted only to those rows that are related to the rows of the table represented by its parent folder. For example the 'Professors' root folder contains all rows from the 'PROFESSORS' table, but the 'Students/Professors' folder contains only those professors who advise at least one student. Similarly, the 'Students/Professors/Classes' folder contains only those classes that are taught by the professors who advise at least one student, and so forth. From this discussion it is obvious that parent folder/subfolder relationships represent inner joins between the tables represented by the two folders, and the contents of a subfolder are the result of a semi-join between the two tables. Later on we will discuss how a parent folder/subfolder relationship can be generalized to represent any binary relational operator such as division, union or full outer join.

There is one problem with our folder-based approach, namely the fact that different instances of the same relation represented as distinct folders will be identically named even though their contents may differ. Although the full path to the folder is an indicator of the folder's contents, ideally each folder should have a descriptive name such as 'Advising professors' or 'Taught classes'. This is impossible to automate without the existence of suitable metadata in the database. Such descriptive naming can however be performed manually by the user by renaming the generated folders. In the future we plan to deal with this problem more thoroughly.

Although seeing that the 'Students' folder has a 'Professors' subfolder reveals that there is a relationship between students and professors, one would be more interested in retrieving data stored in the database regarding these entities, presented in a convenient human-readable form. We hence introduce the concept of a *document* which is a materialized view over the data contained in the document's parent folder and its subfolders. Based on this definition, a 'Names' document in the 'Students' folder representing a vertical view over the STUDENTS table

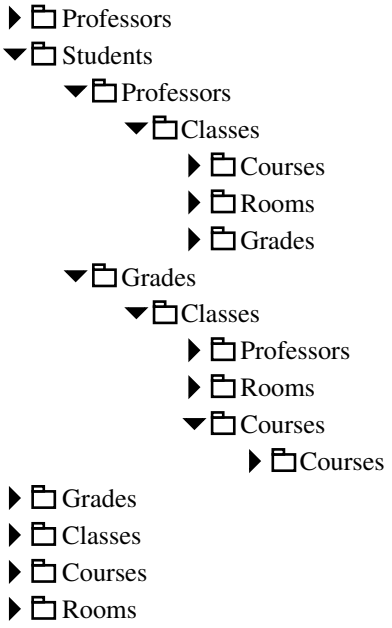

Fig. 3. Partially expanded folder tree representing all possible paths through the Entity-Relationship diagram of figure 2.

on the NAME column would contain the names of all students, whereas a 'Names & Advisors' document referencing the NAME column from the STUDENTS table and the NAME column from the PROFESSORS table would display only the names of those students who have an advisor and the name of that advisor. It follows that such a document could not exist in the 'Student/Professors' subfolder, as the latter only contains information about professors and not students. This highlights a crucial difference between a traditional folder hierarchy, where users can arbitrarily create, copy and move documents between folders, and the QBB paradigm where such actions are restricted based on the contents of the document and the relation represented by the folder.

B. Representing restriction

Suppose that we wanted to find the professors that advise at least one postgraduate student, a query that requires the algebraic operation of *restriction*. This could be achieved by placing a 'Professors' subfolder inside a 'Postgraduate students' folder, the latter being a restricted version of the 'Students' folder. Our initial approach to restriction is similar to that used in the Query by Example query language [3]. We create a special document that we call a *filter* in the 'Students' folder for each column[1] in the STUDENTS table, much like QBE creates headings for all columns in a skeleton table. This is a good way for users to familiarize themselves with the structure of an unknown table. In QBE users may enter literal values in each column to perform comparisons. Multiple values under the same column lead to disjunctions of comparisons. Values across different columns lead to conjunctions

[1]Foreign key columns are excluded.

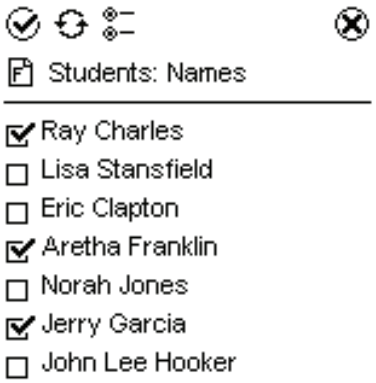

Fig. 4. A student names filter.

of comparisons. This is a very natural way for formulating a number of common queries based on the conjunction of disjunctions.

QBB takes this concept a step further: each one of these column-representing filters displays the unique values of its corresponding column in the table. By using these values to label a group of checkboxes, we allow users to select the multiple values that exist in the database. Selecting more than one checkbox within a filter leads to a disjunction, whereas selecting checkboxes across filters leads to a conjunction. This way users can browse through the contents of a table and query the table at the same time without having to recall the domains and values of individual columns. An example of such a filter is shown in figure 4. For all its obvious ease-of-use benefits, this approach could incur a significant performance penalty when used on large tables. In such cases, each filter could fetch a specific number of distinct values at a time. Caching techniques could also be used to enhance performance perhaps at the expense of accuracy.

An additional enhancement is to continuously update each filter as the user selects values in other filters. This way we avoid the creation of a conjunction that leads to no results, also known as the 'zero hit problem', since any nonexistent combination of values is immediately removed from the user's view. We can also allow the creation of these column-representing filters across multiple columns. In such a case, every checkbox would be labelled using a composite value; selecting multiple checkboxes in such a document thus leads to a disjunction of conjunctions. Thus, by simply creating and interacting with these simple filters a great number of common restriction scenarios can be accommodated.

C. Activation

In QBE it is easy to see which columns have been used in the user's query, as they are the ones containing values in the skeleton table. Similarly in QBB, the filters that have been included in the user's query are clearly indicated in the user interface. We call such filters *activated* and the action of including them in the query *activation*. The opposite of activation is *deactivation*. All filters that have not

been included in the query are considered *inactive*. Such filters might be irrelevant to the query or they might provide contextual information, such as the fact that students have student id's, names, years and levels. By keeping activated and inactive filters within a folder we remain consistent with our goal of unifying schema browsing and querying. This also allows users to explore different scenarios by activating and deactivating filters while observing the effect of a filter on the contents of other filters or documents.

The concept of activation raises some interesting issues when it comes to filters that reference columns drawn not only from the filter's parent folder but its subfolders as well. If the user selects a composite value from such a filter, it follows that there must exist a record in the subfolder that is related to the parent folder, i.e. in selecting a specific student-advisor pair, it is implied that the student has an advisor. Since a parent folder/subfolder relationship represents an inner join (at least for now), then activating such a filter means that the parent folder has to be joined to the subfolder for the filter to have any meaning. In other words, the subfolder too is activated as a consequence of the filter's activation. This 'chain of activation' can in fact extend to any descendant subfolder. For example an activated filter containing student names and the codes of the courses they take would not only activate the 'Courses' folder but also the 'Grades' and 'Classes' folders that relate courses to students. Such a filter would thus contain all student names and the course codes of all classes the student has taken regardless of grade, year and semester.

D. Subfolders as filters

Consider now the case where we want to retrieve the grades of those students who have an advisor. We know from our discussion so far that the 'Students/Professors' folder contains only those professors who advise at least one student. The 'Students' folder however contains all students, not just the ones who have an advisor. Why not then allow the explicit activation of a subfolder to act as a filter for restricting the 'Students' folder only to those students who have an advisor? We have already mentioned that the contents of a subfolder represent a semi-join between the parent folder table and the subfolder table. Activating a subfolder means that the contents of the parent folder table are now the result of the semi-join between the two tables, albeit projected on the parent folder table's columns. Once again by keeping activated subfolders and inactive subfolders within a folder we maintain our approach of unifying schema browsing with querying. Note that this particular problem could also be solved by navigating to the 'Professors/Students' subfolder, which contains only those students who have an advisor. This example illustrates that although QBB uses a hierarchical representation, it does not impose a specific path or order in which queries can be formulated, thus remaining consistent with the relational model.

E. Filters as predicates

Going back to the concept of filters we make the observation that the form we have so far suggested for a filter is in fact similar to that of the IN predicate in SQL, which translates a list of values into the disjunction of a series of comparisons. There is nothing in our discussion of filters so far that precludes the representation of any other kind of predicate as a filter. This is were we think that the QBB paradigm is most extensible and can function as a framework where specialized visual or other querying methods can be incorporated in the form of filters. A GIS system for example could contain a map-based filter. An Alphaslider [4] could be used for filtering long lists of terms using a limited amount of screen space. In fact, any interactive querying technique can function as a filter as long as it produces a Boolean expression, references only the columns of its parent folder or those of its descendant folders and updates its contents immediately based on the user's interactions with other activated objects such as filters and subfolders. The idea of multiple cooperating views on the same data is of course not new [2], yet we believe that the QBB paradigm manages to extend and more formally define the concept.

F. Templates

To increase the re-usability of these querying techniques and filters in general, we introduce the concept of a *template*. A template is essentially a blueprint for creating filters or documents. A template takes table and column names as parameters and produces the appropriate user interface for presenting and interacting with the data contained in these tables and columns in the form of a document or filter. A template for our original concept of a filter for example would take one or more table-qualified column names as parameters and produce a checkbox for every unique combination of values in those columns in the database. Templates are contained in the same folders as the objects they create, again unifying schema browsing with querying and data presentation. The difference between document and filter templates is that the latter must contain the necessary information for generating a Boolean expression based on the user's interactions with the generated filter.

G. Advanced restriction mechanisms

Despite all the advanced restriction mechanisms we have discussed so far, we have still not achieved the expressive power of Boolean algebra. While complex Boolean expressions can be encapsulated in filters and subfolders, users have so far no option of combining these expressions other than conjunction. To this end we define the *criteria combination option*. This is a declarative way for combining the criteria of all activated objects in a folder that is as powerful as Boolean algebra, yet easier to read, verify and modify than a Boolean expression. It consists of a comparison operator ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$), a positive integer or the special keyword 'all' and a choice between the option 'satisfied' or 'failed'. Together these define the number of activated filters or subfolders whose conditions need to be satisfied or failed by a row in the table represented

by the folder. For example the specification '= all satisfied' leads to a conjunction while specifying '≥ 1 satisfied' leads to a disjunction. A formal definition for the criteria combination option is given in section II-L.

H. Grouping folders

A feature that is commonly missing from visual query languages is the ability to parenthesize Boolean expressions in order to change the precedence of logical operators. QBB supports this through the introduction of *grouping folders*. These folders do not represent a relation but are used solely for grouping subfolders, documents, filters and templates. In this capacity they can be thought of as equivalent to folders in a traditional folder hierarchy. Grouping folders however can be activated just like any other folder and through their criteria combination option they combine the criteria of all activated objects they contain. For example consider a folder f representing a table T with a criteria combination option of '= all satisfied' and containing two activated grouping folders both with criteria combination option '≥ 1 satisfied'. Assuming that each grouping folder contains two activated filters representing predicates A, B and C, D respectively, then the contents of f are $\sigma_{(A \vee B) \wedge (C \vee D)}(T)$. Such an expression would be impossible to formulate without the use of grouping folders.

I. Representing binary relational operators

All binary relational operations are represented in QBB through parent folder/subfolder relationships. The relation represented by the parent folder is the first or left operand, while the relation represented by the subfolder is the second or right operand. A new binary relational operation is performed by creating a new subfolder. In order to streamline the process of creating a new subfolder and hence enhancing ease-of-use, we have made a number of decisions. First, we divide all binary relational operators into two families: set operators and join operators. The first category is self-explanatory. The second category includes not only the various join operators but also the division and recursive join operators, the latter described in [5]. Building on Date's definition of the generalized relational division operator [6], we extend division to allow the use of any condition and the inclusion of any subset of the divisor and dividend's columns in the quotient. Finally, we support the outer set operators, again as defined by Codd [5], thus relaxing the restriction for union compatibility between the relations represented by the parent folder and the subfolder. Thus our streamlined subfolder creation process begins as follows:

The user navigates to the folder (e.g. 'Students') creates a new subfolder, and provides a short descriptive name for the new folder. There are then two possibilities: either the new folder is a grouping folder or it represents a table or view in the database. In the former case, no further actions are needed, other than of course creating, copying or moving activated filters and subfolders inside the grouping folder and setting its criteria combination option. In the latter case, the table or view that will serve as the basis for the folder's contents (e.g. 'PROFESSORS'), must first be selected. Finally, the template that will be used to populate the folder with an initial set of objects must be selected. The standard QBB template provides the schema-compliant view that we have presented so far, however other templates can be used to customize the user's view over the database schema, as will be discussed in section II-K. This first step at creating a subfolder is shown in figure 5.

Fig. 5. First step at creating a new subfolder. If the 'Contains' checkbox is left unchecked, then we have a grouping folder. The 'Professors' link invokes a page allowing the user to select a different database table.

1) Join operators: In the second step, the user must decide whether the two folders will be cross-referenced (join operation) or combined (set operation). If the 'cross-reference' option is selected then we search the database schema for any primary/foreign key constraints between the tables represented by the two folders, or between the tables and intersection tables that represent many-to-many relationships. These constraints are presented in a list allowing the user to select one or more of them. If no primary/foreign key constraints exist between the two tables, then the user can manually enter the condition that is used to join them. This condition is formulated using filters, subfolders, grouping subfolders, shortcuts and the criteria combination option, in other words using the full expressive power and schema browsing capabilities of the QBB paradigm. The same interface can be used to edit any conditions that have been generated by QBB based on primary/foreign key constraints.

Once the join condition is specified, the user needs to decide whether this condition should hold true for at least one record in the subfolder or for all records in the subfolder. The former of course leads to a join between the two folders whereas the latter leads to a division. We in fact offer a more advanced option that allows the user to specify any range of records in the subfolder that need to satisfy the join condition. This allows the easy and declarative formulation of queries such as 'find all students who are taught by two professors named John', without having to resort to the use of grouping (GROUP BY), aggregate functions (COUNT) and filtering of groups (HAVING). It also leads to a more natural and consistent interface due to its similarity to the criteria combination option.

The final step in cross-referencing the two folders is to

specify whether the records from either the parent folder or subfolder or both that do not satisfy the join condition should be included in the result set as well. We call this the *inclusion option*. If the user selects that all records from the parent folder should be included then we have a left outer join or division. If the user selects that all records from the subfolder should be included then we have a right outer join. If the user selects that all records from both folders should be included then we have a full outer join.

2) Recursive join: There is a special case when cross-referencing two folders that represent the same table. In such a case, we could potentially have a recursive query as long as the following conditions are satisfied:

- The two folders are related using an inner join.
- At least one of the following applies:
  - The two folders are related through an intersection table. Each folder then represents a set of nodes in a graph.
  - The two folders are related directly and the set of referenced columns in the join condition drawn from the parent folder table is distinct from the set of columns drawn from the subfolder table. Each folder then represents a set of nodes in a hierarchy (tree).

We then ask the user whether the join condition should be applied recursively. If so, then QBB transparently applies the recursive join operator previously described in order to find all possible paths through the graph or hierarchy. If the two folders are related through an intersection table then that intersection table becomes the operand of the recursive join operator. The SUPerior columns are those columns drawn from the intersection table that are joined to the parent folder, whereas the SUBordinate columns are those that are joined to the subfolder [2]. If the two folders are related directly then the table they represent becomes the operand of the recursive join operator. The primary or candidate key columns of the table become the SUPerior columns and the remaining columns become the SUBordinate columns[3].

Once all the paths have been discovered using the recursive join operator, we need to establish the relationship between the node set represented by the parent folder and that represented by the subfolder. We have modelled our recursive querying capabilities after the XPath 2.0 query language. Every folder in a recursive query represents an axis step. The output of every axis step becomes the input to the next axis step until the desired set of nodes is retrieved. The user is given a set of mutually exclusive options, each one representing an XPath 2.0 axis, originally defined for XML document trees, and redefined here to encompass graphs as well. These options are:

- Child: for every subfolder node there exists an *arc* to at least one parent folder node (distinct from the subfolder node) that is directed from the parent folder node to the subfolder node.
- Parent: for every subfolder node there exists an *arc* to at least one parent folder node (distinct from the subfolder node) that is directed from the subfolder node to the parent folder node.
- Descendant: for every subfolder node there exists a *path* to at least one parent folder node (distinct from the subfolder node) that is directed from the parent folder node to the subfolder node. The descendant axis is the transitive closure of the child axis.
- Ancestor: for every subfolder node there exists a *path* to at least one parent folder node (distinct from the subfolder node) that is directed from the subfolder node to the parent folder node. The ancestor axis is the transitive closure of the parent axis.
- Following: every subfolder node has a greater pre-order position[4] than at least one parent folder node, of which it is not a descendant.
- Preceding: every subfolder node has a smaller pre-order position than at least one parent folder node, of which it is not an ancestor.
- Following sibling: every subfolder node has a greater pre-order position than at least one parent folder node, with which it has at least one common parent node.
- Preceding sibling: every subfolder node has a smaller pre-order position than at least one parent folder node, with which it has at least one common parent node.

The last four axes clearly require that the nodes in the hierarchy be sorted so that their preorder position can be determined. To this end we ask the user to provide a sort specification, which is an ordered set of column name-ordering direction(ascending, descending) pairs. To guarantee the repeatability of this process there must be no ambiguity about the sorting order of a row which implies that the combination of values of the selected sorting columns must be unique for every row. All this information can be saved under a unique name and retrieved next time a recursive query over the same relation is required.

Suppose for example that we want to find all prerequisite courses for a course with code 'CS101'. We first begin with a root folder representing the 'COURSES' table which we name 'CS101'. We then restrict the rows in this folder using a filter projecting 'COURSES' on the 'Code' column and selecting the value 'CS101'. We create a subfolder representing the 'COURSES' table inside the 'CS101' folder which we name 'Prerequisites'. We select the inner join operator to relate the two folders and we select the suggested join condition that uses the 'COURSES_PREREQUISITES' table. We request that the join condition be applied recursively and we select the 'descendant' option. Unlike the automatically generated 'Courses' subfolder that contains the direct prerequisites of course 'C101', the 'Prerequisites' subfolder contains the transitive closure of all its prerequisite courses. Such a query would be impossible to express using first-order logic.

[2]This means that the two folders must be distinctly named.

[3]Recall that a hierarchy is a recursive 1:N relationship.

[4]The pre-order position of a node is the order in which it is encountered in a pre-order traversal of the hierarchy or network.

3) Set operators: In case the 'combine' option is selected then we check whether the two tables represented by the parent folder and the subfolder are union compatible. If so, then we suggest a set of possible pairings of columns between the two tables[5]. Otherwise, or after selecting the suggested set of pairings, the user can specify pairings between columns from the two tables drawn from the same domain or that can be converted to a common domain using type conversion functions. The user then selects one of the three set operators (UNION, INTERSECTION and DIFFERENCE) that will be used to combine the records of the two folders. Any document in the parent folder that references only the paired columns of the two folders uses the inner version of the set operators, whereas any document that references at least one non-paired column uses their outer versions.

J. Manipulation, grouping, sorting and aggregation

We have so far not discussed the manipulative operators *insert*, *update* and *delete*. These are supported through documents that present the appropriate interface for creating, deleting and updating records. Since documents are views over the contents of their parent folder and its subfolders which in turn are views over the database tables (or views) they represent, the usual considerations regarding view updatability apply. For reasons of brevity we will not discuss these further.

Another important operator that we have not yet discussed is the *framing* operator. In order to explain our approach to framing, we first need to describe our approach to attribute creation which touches on a number of different areas. Within a folder, users are given the option of creating new attributes that apply to all records in the folder. There are five different types of user-defined attributes:

- *Calculated attributes* are based on expressions involving literal values, folder attributes, operators and functions. A calculated attribute for example could involve a function that translates a numeric student grade into its equivalent letter grade.
- *Composite attributes* are the combination of two or more existing attributes that are treated as a unit. For example the 'Course_code', 'Semester' and 'Year' attributes can be combined into a single class identifier attribute. Whenever a composite attribute appears in a logical expression, it is treated as the conjunction of the values of its constituent attributes.
- *Categorizing attributes* are drawn from a discreet domain and assign a unique value to a group of records based on the values of one or more of their attributes. For example we can categorize students by the year they were enrolled in.
- *Ranking attributes* are based on a sort specification involving attribute name-sorting direction (ascending, descending) pairs. They thus assign a unique value to every record in the folder. Such an attribute could be used for ranking students within a specific class based on their grade. Ranking attribute specifications can optionally reference a categorizing attribute to assign a unique value to every record within a group.
- *Summarizing attributes* involve an aggregate function such as SUM, AVG and COUNT. An optional categorizing attribute can be specified, allowing the calculation of summary data for each group of records rather than all records in the folder.

Once these attributes have been created, they can be referenced in all documents, filters, join conditions and column pairings in the folder. In order to group or sort the contents of a document or filter, the *grouping* and *sorting directives* need to be used. These are identical to the GROUP BY and ORDER BY clauses in SQL and need not be discussed further. These directives are invoked interactively by the user through some suitable user interface, and are used for presentation purposes only. By recording ordering and grouping information explicitly in ranking and categorizing attributes rather than implicitly in the way the result set is presented, we are able to create multiple rankings and groupings of records which can also be used in logical or arithmetic expressions. This obviates the need for an SQL-style HAVING clause for filtering groups of records. It also ensures that we abide by Codd's 'information feature' [5], which requires that all *essential* ordering information be explicitly recorded in relations.

K. Folder templates

While most visual query languages follow a rigid join-centric approach to schema browsing, such as the one we have discussed so far, QBB takes schema browsing to the next level by allowing the creation of flexible views over the database schema using the full expressive power of the QBB paradigm. This is achieved is through folder templates. A template is a parameterized query that can be thought of as equivalent to a dynamic SQL statement. A template can query both the database's intention and the database's extension. Normally the database's intention contains descriptions of all tables, views, domains, constraints, functions and triggers in the database in the form of tables known as the system catalog. In QBB we extend this to encompass descriptions of all folders, documents, filters, shortcuts and templates.

A folder template is different from document or filter templates in that it may create new QBB objects to populate the folder it creates. For example a folder template for the type of folder we have so far described would query the primary/foreign key constraints in the database and create a new subfolder for every table related to the folder table through such a constraint. It would also create an IN predicate-style filter for every column in the folder table. Another type of template could create a document for every row in the folder table containing all columns and named using the (composite) value of the table's primary key. Different folder templates could allow for different views over the data contained in the table represented by the folder, the tables that are related to it, and the QBB objects that can be used to query the table.

[5]For reasons of brevity, we do not discuss the details of the column-pairing algorithm.

L. Formal definition

Tuple relational calculus, thereon referred to simply as relational calculus, is the formal basis of the Query by Browsing paradigm. This means that, like relational calculus, QBB is descriptive rather than prescriptive. A relational calculus expression is represented using the form $\{t|P(t)\}$ where t is a tuple variable and $P(t)$ is a well formed formula used for restricting the values of t. We use the notation $t \in R$ to indicate that the range of tuple variable t is the relation R.

A folder f represents a relation B_f that we call the folder's *base relation*. The data content of the folder, D_f is a relation such that $D_f \leftarrow B_f(C_f())$, where $C_f()$ is a well formed formula used of restricting the contents of D_f which we call the folder's *folder criteria*. This means that D_f is a horizontal view over B_f. Initially $C_f()$ might be empty, in which case $D_f \leftarrow B_f$.

For a *root folder*, $C_f()$ is of the form $t \in B_f\,[\{N\}]\,[satisfied|failed]\,(A_1(t), A_2(t), \ldots, A_n(t))$, where t is a tuple variable ranging over B_f, N is an optional set of positive integers and $A_1(t), A_2(t), \ldots, A_n(t)$ are well formed formulae represented by activated objects in f. The meaning of this notation is as follows: return all tuples t such that the number of formulae in the list $A_1(t), A_2(t), \ldots, A_n(t)$ that either return true (satisfied) or false (failed) c is such that $c \in N$. If the specification for N is omitted then $N \leftarrow \{n\}$, where n is the number activated object formulae. If the words 'satisfied' or 'failed' are omitted, then 'satisfied' is assumed. Thus the notation $t \in B_f(A_1(t), A_2(t), \ldots, A_n(t))$ is equivalent to $t \in B_f(A_1(t) \wedge A_2(t) \wedge \ldots \wedge A_n(t))$. Similarly, $t \in B_f\, failed\,(A_1(t), A_2(t), \ldots, A_n(t))$ is equivalent to $t \in B_f \neg(A_1(t) \vee A_2(t) \vee \ldots \vee A_n(t))$. The notation $t \in B_f\{x \geq 1\}(A_1(t), A_2(t), \ldots, A_n(t))$, where x is a dummy variable that can be omitted, is equivalent to $t \in B_f(A_1(t) \vee A_2(t) \vee \ldots \vee A_n(t))$. Finally, the notation $t \in B_f\{\geq 1\}\, failed\,(A_1(t), A_2(t), \ldots, A_n(t))$ is equivalent to $t \in B_f \neg(A_1(t) \wedge A_2(t) \wedge \ldots \wedge A_n(t))$. Any other case would result in the disjunction of at most 2^n n-term conjunctions, each one containing every formula $A_x(t)$ exactly once, either as is $(A_x(t))$ or inverted $(\neg A_x(t))$.

For a subfolder, $C_f()$ is of the form $t \in B'_f\,[\{N\}]\,[satisfied|failed]\,(A_1(t), A_2(t), \ldots, A_n(t))$ where $B'_f \leftarrow B_f(S_f())$, where $S_f()$ is a formula restricting the subfolder's data content based on its relationship to its parent folder as defined using a formula $R_f()$. Table I shows the relationship between the relational operator represented by the parent folder/subfolder relationship, the corresponding formula $R_f()$ and the subfolder restricting formula $S_f()$. In general, the data content of a folder is given by the formula $f \in B_f(S_f() \wedge C_f())$; $S_f()$ for root folders is set to 'true'.

Every document or filter contains one or more queries of the form $\{d_1[A_1], d_2[A_2] \ldots d_n[A_n] || d_1 \in D_{f_1}(F_d(d_1, d_2 \ldots d_n))\}$ where $n-1$ is the number of subfolders whose attributes are referenced in the query. D_{f_1} is the data content of the document or folder's parent folder, while d_1 is a tuple variable ranging over D_{f_1} and A_1 is the set of parent folder attributes projected by the document or filter. Similarly, d_i where $2 \leq i \leq n$ is a tuple variable ranging over the data content of a subfolder D_{f_i} and A_i is the set of projected attributes from that subfolder. Recall that $D_{f_i} \leftarrow B_{f_i}(C_{f_i}())$. $F_d()$ is a formula that further restricts the contents of a document or filter with respect to the data content of its parent folder D_{f_1}. This formula involves the formulae $R_{f_i}()$ of all subfolders whose attributes are either projected in the document, or used for restricting the content of the document. The fact that in the case of activated subfolders these formulae may also be included in a folder's folder criteria $C_{f_i}()$ does not affect the result of the query since all formulae are idempotent (i.e. $f(x) = f(f(x))$).

Filters of course in addition produce a formula $F_f()$ that is used for restricting the data content of their parent folders when activated. Templates on the other hand contain the same type of queries as documents. The only difference is that the tuple variables in those queries may range over relations that describe the database's intention, and may also include 'parameter markers' for relation and attribute names.

TABLE I

CORRESPONDENCE BETWEEN THE RELATIONAL OPERATOR REPRESENTED BY A FOLDER/SUBFOLDER RELATIONSHIP, ITS CORRESPONDING RELATIONAL CALCULUS FORMULA AND THE FORMULA RESTRICTING THE CONTENTS OF THE SUBFOLDER. THE DATA CONTENTS OF THE PARENT FOLDER AND THE SUBFOLDER ARE D_f AND D_s RESPECTIVELY, WHILE $J(s, f)$ IS THE JOIN CONDITION RELATING THE TWO FOLDERS.

Relational operator	$R_f()$	$S_f()$
$\cup$	$f \in D_f \vee f \in D_s$	$s \in D_s - (D_f \cap D_s)$
$\cap$	$f \in D_f \wedge f \in D_s$	$s \in D_f \cap D_s$
$-$	$f \in D_f \wedge \neg f \in D_s$	$s \in D_f \cap D_s$
$\bowtie$	$\exists s \in D_s(J(s, f))$	$s \in D_s(\exists f \in D_f(J(s, f)))$
$\div$	$\forall s \in D_s(J(s, f))$	$s \in D_s$

M. Example

The base relation of the 'Professors' root folder is the 'PROFESSORS' table. Since it is a root folder, and assuming that there are no activated objects in the folder, its data content is $p \in PROFESSORS$. The 'Students' subfolder is related to the 'Professors' folder through a join and hence the formula $\exists s \in STUDENTS(s[Advisor] = p[Faculty\_ID])$. The data content of the 'Students' subfolder, again assuming that no activated objects exist in that folder, is $s \in STUDENTS(\exists f \in PROFESSORS(s[Advisor] = p[Faculty\_ID]))$.

Let us now create two documents, 'Names' and 'Advised students'. The former contains the names of all professors in the folder while the latter contains the names of all professors and the names of the students they advise. The 'Names' document will contain the query $p[Name] || \{p \in PROFESSORS\}$ while the 'Advised students' document will contain the query $p[Name], s[Name] || \{p \in PROFESSORS(\exists s \in STUDENTS(s[Advisor] = p[Faculty\_ID]))\}$. In the first document $F_d()$ is empty, while in the second document $F_d() = \exists s \in STUDENTS(s[Advisor] = p[Faculty\_ID])$.

If now we activate the 'Students' subfolder, the data content of the 'Professors' folder changes to $D_f \leftarrow PROFESSORS(\exists s \in STUDENTS(s[Advisor] = p[Faculty\_ID]))$. The 'Names' document now contains

the query $p[Name]|\{p \in PROFESSORS(\exists s \in STUDENTS(s[Advisor] = p[Faculty\_ID]))\}$, as the folder criteria formula $C_f()$ for the 'Professors' folder is now $C_f() = \exists s \in STUDENTS(s[Advisor] = p[Faculty\_ID])$. The content of the 'Advised students' document however is not affected since its query now becomes $p[Name], s[Name]|\{p \in D_f(\exists s \in STUDENTS(s[Advisor] = p[Faculty\_ID]))\}$ which is equivalent to $p[Name], s[Name]|\{p \in PROFESSORS(\exists s \in STUDENTS(s[Advisor] = p[Faculty\_ID]) \wedge \exists s \in STUDENTS(s[Advisor] = p[Faculty\_ID]))\}$. Since $A \wedge A = A$, this query is reduced back to the original document query.

III. Expressive power

From our discussion so far it should be clear that the QBB paradigm supports all relational algebra operators. Projection (π), insert, update and delete are supported through documents, filters and user-defined attributes. Restriction (σ) is supported through filters, grouping folders and the criteria combination option. All binary relational operators including union ($\cup$), intersection ($\cap$), difference ($-$), join ($\bowtie$) and division ($\div$) and their outer versions as well as recursive join ($\bowtie^*$) are supported through parent/subfolder relationships created using the streamlined declarative process described in section II-G. Framing is supported through categorizing attributes and the grouping directive. Likewise, sorting is supported through ranking attributes and the ordering directive. In fact, the only algebraic operators not supported by QBB are T-join and outer T-join. We believe that such extensive coverage of relational algebra in a visual query language is unprecedented.

IV. Usability

One of the strongest points of QBB is its similarity to the desktop user interface paradigm. In fact there are only three concepts that QBB introduces that do not exist in a typical file browser:

- Filters
- Activation
- The criteria combination option

However, equivalent concepts are starting to appear in modern operating systems, particularly in the next versions of Mac OS X [7], code named 'Tiger', and Windows [8], code named 'Longhorn'. In these interfaces filters are restricted to simple comparisons and are always activated, while the criteria combination option is implicitly a conjunction (i.e. 'all criteria satisfied'). We believe that our approach to restriction is a natural extension to these simple search interfaces as its declarative nature, compactness and modelessness makes it compatible with basic desktop paradigm principles [9].

From our initial discussion on filters the similarity between QBB and QBE should be evident. The usability characteristics of QBE are well documented, so we needn't delve into this issue further. Again we consider some of the QBB restriction techniques to be the natural evolution of the QBE paradigm. Skeleton tables have been replaced by folders, headings have been substituted with filters and rather than having to type constant elements under column headings, the user can select from a list of actual values from the database. Example elements used for joining two skeleton tables are represented using parent folder/subfolder relationships, which can also be automatically generated using folder templates. Folder templates can also be used to create filters or documents for every column in the table represented by a folder, much like QBE can automatically generate the skeleton table headings. It seems then that in some ways QBB can be seen as the modernization and expansion of the QBE paradigm that inherits its ease-of-use characteristics while contributing its own.

With its approach to filters and documents as multiple simultaneous views over the same data, QBB inherits many of the well-documented advantages of dynamic queries and information visualization [10]. These includes the avoidance of the 'zero hit' or 'too many hits' problems, the encouragement of data exploration and the simultaneous maintenance of focus and context. By emphasizing schema browsing and going a step further, by allowing the creation of custom views over the database schema using folder templates, QBB incorporates and enhances one of the most cited advantages of visual query languages. With its unified approach to schema browsing, querying and presentation of results, it avoids unnecessary modes and standardizes on a minimal set of objects, actions and conventions. Its streamlined approach to the representation of binary relational operations also contributes to its elegance. Finally, its declarative rather than prescriptive approach to logical expressions, join and division leads to easier to understand, verify and modify queries.

Our ease-of-use expectations have so far been corroborated by informal user testing results. While there is a learning curve for QBB, it is considerably short for simple tasks and comparably smoother than SQL's for more complicated tasks. In the near future we plan to run more detailed and formal usability tests and to incorporate the feedback received from these tests to improve the usability of QBB further.

V. Flexibility

While QBB is heavily based on the relational model, it can accommodate other data models including the entity-relationship and object-oriented models, and the XML Infoset. Table II lists the equivalences between QBB concepts and concepts from these models.

The relationship to the Entity-Relationship model has already been discussed in section II-A. The XML Infoset produces a structure-based rather than content-based decomposition of an XML document into element, attribute, namespace, processing instruction, comment and other nodes. Each type of node can be considered a mutually exclusive subtype of a generic node type. A folder can be used to represent a set of nodes, with a subfolder for each type of node, e.g. 'Elements', 'Attributes' and so forth. Element folders contain namespace, element and attribute subfolders. Using QBB's recursive query capabilities we can achieve the same expressive power as the XPath and XQuery XML query languages.

Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)
1084-4627/05 $20.00 © 2005 IEEE

TABLE II

APPLICATION OF THE QBB PARADIGM TO NON-RELATIONAL SCHEMATA.

Model	Concept	QBB equivalent
Entity-relationship	Entity	Folder
	Relationship	Folder containement
	Relationship (with attributes)	Subfolder
	Attribute	Attribute
	Entity instance	Record
Object-oriented	Class	Folder
	Object	Record
	Attribute	Attribute
	Method	'Method' subfolder
	Class hierarchy	Folder hierarchy
XML Infoset	Node set	Folder
	Node containment	Folder containement
	Node	Record

VI. SCALABILITY

Throughout this paper we have demonstrated the scalability of our proposed paradigm by including screenshots from *Chiromancer*, a web-based implementation of the QBB paradigm for mobile devices. It is to our knowledge the *only* visual query language for such a form factor. The QBB paradigm is scalable in two ways: it has a very small screen footprint, and can accommodate large schemas.

The former is achieved partly due to the unified approach to schema browsing, querying and presentation. This means that no separate modes exist for these tasks as this would require a dedicated area in the user interface to switch between modes and to inform the user of the currently active mode. The criteria combination option also reduces screen requirements dramatically since logical operators do not have to be listed in the interface. The savings increase exponentially as the complexity of the logical expression increases. Last but not least, the representation of the schema as a folder hierarchy, rather than the more common diagrammatic approaches employed in other visual query languages, requires significantly less screen space by substituting the spatial juxtaposition of tables and columns with a temporal one through scrolling, selective disclosure and folder navigation.

Accommodation of large schemas is achieved of course primarily through QBB's minimal screen requirements. The hierarchical representation of the schema also scales better than diagrammatic approaches as the number of tables and columns increases. Moreover, through folder templates, more specialized and compact views over the database schema can be constructed, by hiding irrelevant tables, columns and relationships from the mobile user's view. However, the possibility of ending up with too many folders or too many documents and filters does exist. In such a case, popular file browsing techniques that deal with this problem, such as searching for documents and folders using simple keyword search on their names or content (i.e. data values) could be used.

VII. IMPLEMENTATION

A prototype implementation of QBB has been developed using IBM DB2 UDB v8.1, PHP 5.0 and the Apache 2 web server. Before a DB2 database can be queried using QBB, it has to be *enabled* for QBB. This means that all the database objects required by QBB must be created on the database. These include the 'QBB' schema as well as a number of user-defined types, tables, views, stored procedures, user-defined functions and triggers. This decision was made primarily for performance purposes. QBB requires the maintenance of data structures describing folders, documents, filters, templates and user-defined attributes. By keeping these structures in the database in the form of tables, we are able to centrally manage them and have them constantly updated using triggers. This also makes QBB self-describing. By encapsulating the application logic in triggers and stored procedures, we are also able to access and present the QBB interface from a variety of clients based on different technologies. Note that the QBB folder hierarchy is not created a priori; only the contents of those folders, documents and filters that have been opened at least once by the user are generated.

PHP is used for implementing the initial script that enables a database for QBB. It is also used as the presentation technology for the QBB interface including folder, document and filter contents. Communication with the database both for retrieving the results of queries and the descriptions of QBB objects is done through PHP's unified ODBC function calls. These are actually mapped to the DB2 CLI calls, thus incurring a minimum performance penalty. PHP is used as an Apache 2 module, which means that all processing and presentation is performed on the server, leading to very thin clients, namely a simple web browser. This approach serves our plans for supporting handheld devices with limited processing power and storage space. In the future we might investigate richer client architectures as well as caching techniques to better balance the load and bandwidth between the client and the server in order to improve latency, reliability and response time to user actions.

We have also implemented an a structure-based XML 'shredder' that decomposes an XML document into relational tables according to the XML Infoset using PHP's XML parsing capabilities. QBB can then be used on these tables to query XML documents using the full power of relational algebra. This is an area where the recursive querying capabilities of QBB are most applicable. The implementation of these capabilities deserves a section in and of itself, but for reasons of brevity we will not discuss it in detail. Information about a graph is created using a stored procedure that creates and populates several tables, including a table describing all possible paths through the graph and another one containing information about all nodes in the graph. These tables are again kept up-to-date using triggers. The procedure takes the following parameters: the name of the table representing an arc, in the case of a graph, or a node, in the case of a hierarchy, the names of the SUPerior columns and those

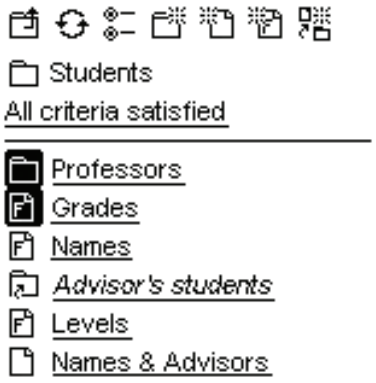

Fig. 6. A typical Chiromancer screen.

of the corresponding SUBordinate columns, and the names and ordering directions of the ordering columns. Once all graph-describing tables are populated, a set of user-defined functions that query these tables are used to determine for example whether a node is the descendant of another node, or whether two nodes are siblings. These UDFs are used in a join condition when cross-referencing two folders, in order to implement the axes we have defined in section II-I.2.

VIII. Chiromancer II

Chiromancer II is a web-based implementation of the QBB paradigm for handheld devices. It is to our knowledge, as of the time of this writing, the only relationally complete visual querying interface for handheld devices. Chiromancer adopts a one-screen-at-a-time approach and relinquishes horizontal scrolling. We believe that both these points are important in producing a usable interface for handheld devices. Thus users can only see the contents of one folder, filter or document at a time, in a vertically scrolling format.

Figure 6 shows a typical Chiromancer II screenshot. The icon bar at the top allows the user to navigate to the parent folder, to refresh the page displaying the contents of the folder or document, to change presentation options for the currently displayed object such as sorting or grouping the contents of the displayed folder, filter or document and to create new subfolders, documents, filters and shortcuts (not discussed in this paper). Next comes the name of the currently displayed folder followed by the criteria combination option, shown as a link to a page that allows the user edit its definition. The actual contents of the folder are shown below. In this particular example we see an activated subfolder and an activated filter, followed by inactive filters, folder shortcuts and documents. In other parts of the interface, shown in figures 4, 5, 'OK', 'Cancel', 'Previous' and 'Next' icons are used, always appearing consistently at the same position.

All information is shown in a vertical layout, so the user never has to resort to horizontal scrolling. All Chiromancer pages make judicious use of graphics and are monochrome. No mouse over effects or popups are used; clearly indicated links which are always underlined are used to launch dedicated pages for each task such as opening a folder or document or changing the criteria combination option. The icons employ a simple and consistent visual language based on well-established representation conventions. All these design decisions contribute towards the usability and scalability of Chiromancer and create a minimal set of requirements on the client device with respect to display and browser capabilities, thus making it suitable for use on most handheld devices.

IX. Related work

We have already discussed the similarities between QBB and QBE [3], that extend beyond the choice in naming. QBB adapts many of the ideas found in QBE to the modern WIMP interface. Unlike QBB however, QBE does not offer any schema browsing capabilities other than automatically listing the columns of a table in the form of headings, nor does it offer any assistance when it comes to joining two tables. QBE resorts to a condition box and Boolean algebra for advanced restriction, while QBB maintains a declarative interface that is integrated with the browsing and querying area. QBE like QBB is one of the few VQLs that supports division, through 'set links'. QBE however has the advantage of supporting some DDL tasks such as table creation. While this task is not beyond the representation capabilities of QBB, it is a subject that we have not dealt with yet and plan to in the future.

PESTO [11] is another VQL that closely resembles QBB, although the former was designed with object-oriented databases in mind. The most important similarity is PESTO's unified approach to querying and schema browsing, which its authors call 'query-in-place'. PESTO also supports 'filtered browsing', a concept that is not unlike our approach to activating objects. In PESTO a class of objects is represented through a window that can be used to display individual object instances of the class. In QBB this is achieved by creating a document, an approach which has the added advantage of allowing the customization of how the information in the folder is shown as well as allowing multiple views over the same data. A PESTO window can be put into 'query mode', in which case users can type predicates on the displayed fields, before switching back to 'browse'mode' to observe the results of the query. QBB is more flexible in this area as restriction is achieved through a compact and powerful declarative interface that does not resort to Boolean algebra. Moreover, it can accommodate specialized restriction techniques in the form of filters for special cases. Some of the limitations of PESTO, not shared by QBB, are the lack of support for projection, union, sorting, grouping and aggregation. But perhaps the single most important advantage of QBB over PESTO is its scalability; PESTO follows a more conventional diagrammatic approach to schema browsing which as we have already argued is not very scalable and therefore unsuitable for small screen form factors.

Query By Icon [12] is one of the few VQLs, with the exception of QBB, that claims to be applicable to mobile devices. The authors of QBI make this claim due to the fact that the QBI interface requires no typing, no special knowledge of a remote database's schema and assists in the

formulation of queries without accessing the actual data in the database, which saves precious bandwidth and therefore cost as well as being more resistant to network interruptions. This latter area is perhaps one of the weaker points of QBB. However as we explore caching techniques and richer clients in the future we may be able to alleviate this problem. QBI seems to be much more limited in terms of the queries it can express and the ways in which the results of those queries are presented. Moreover, it has significantly greater screen space requirements than QBB, as it consists of three separate windows: the 'Workspace', the 'Query space' and the 'Browser'. This means that QBB is to our knowledge the only VQL actually designed with handheld devices in mind.

Young and Shneiderman [13] have suggested a very interesting and successful graphical method for representing Boolean expressions that deserves to be mentioned here as a yardstick for the usability and expressive power of the restriction techniques provided in QBB. Their technique consists of a series of graphical filters connected either 'in series' or 'in parallel' to represent conjunction and disjunction respectively. Being a dynamic query interface, the displayed result set becomes updated as each filter is modified by the user. The similarities to QBB are apparent, however Young's filter technique again requires significantly more screen space than QBB. Moreover, it closely follows the structure of Boolean algebra, with all the aforementioned problems that such a decision entails.

There are of course a great deal more VQLs in the research literature [14], [15], [16], [17], [18], [19] and in commercial products. It would be impossible to discuss all of them within the scope of this paper. In general however, these are the areas in which most of these interfaces lag with respect to QBB:

- Lack of support for set operators
- Lack of support for division
- Lack of (graphical) support for restriction that is equivalent to Boolean algebra
- Lack of recursive query support
- Diminishing returns as schemata become more complex
- Screen requirements that exceed the capabilities of handheld devices
- Specific to a 'data model' (relational, entity-relationship, functional, semantic etc)

We have also not discussed the relative benefits of QBB over SQL, although that will be the subject of a future comparative usability study.

X. Conclusion

In this paper we introduced the QBB paradigm, a visual query language that is relationally complete, scalable enough for handheld device form factors, flexible enough to accommodate a variety of different data models and inherently usable due to its close similarity to the popular desktop user interface paradigm. We believe that each of these salient characteristics of QBB constitute it a unique and revolutionary approach to search interfaces.

In the future we plan to run a comparative usability study between QBB and SQL, as well as QBB and other visual query languages and search interfaces such as QBE and Microsoft Access. We also want to enhance support for XML data sources and conduct a similar comparative study between QBB and XPath/XQuery. A richer, more interactive desktop version of QBB is in plan, as is a comprehensive metadata framework that would allow us to automatically provide descriptive names to QBB folders. Finally, while we believe that QBB is one of the most complete visual query interfaces ever conceived, we want to extend it further to encompass DDL tasks such as table and view definitions.

References

[1] C. Batini, T. Catarci, M. F. Costabile, and S. Levialdi, "Visual query systems: A taxonomy," in *VDB*, 1991, pp. 153–168. [Online]. Available: citeseer.ist.psu.edu/batini92visual.html

[2] C. Ahlberg and B. Shneiderman, "Visual information seeking: tight coupling of dynamic query filters with starfield displays," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Press, 1994, pp. 313–317.

[3] M. M. Zloof, "Query-by-example: a data base language," *IBM Systems Journal*, no. 4, pp. 324–343, 1977.

[4] C. Ahlberg and B. Shneiderman, "The Alphaslider: A Compact and Rapid Selector," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Press, 1994, pp. 365–371.

[5] E. F. Codd, *The relational model for database management: version 2*. San Francisco, CA: Addison-Wesley, 1990.

[6] C. J. Date, *An Introduction to Database Systems 5th Edition*. San Francisco, CA: Addison-Wesley, 1991.

[7] Apple Computer Inc.", "Mac OS X," www.apple.com.

[8] Microsoft Corporation, "Microsoft Windows XP," www.microsoft.com.

[9] Apple Computer Inc.", *Macintosh Human Interface Guidelines*. Addison-Wesley Publishing Co., 1992.

[10] S. K. Card, J. D. Mackinlay, and B. Shneiderman, *Information Visualization: Using Vision to Think*. San Francisco, CA: Morgan Kaufmann Publishers, 1999.

[11] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams, "PESTO : An integrated query/browser for object databases," in *The VLDB Journal*, 1996, pp. 203–214. [Online]. Available: citeseer.ist.psu.edu/haas96pesto.html

[12] A. Massari, S. Weissman, and P. K. Chrysanthis, "Supporting mobile database access through query by icons," *Distributed and Parallel Databases*, vol. 4, no. 3, pp. 249–269, 1996.

[13] D. Young and B. Shneiderman, "A graphical filter/flow representation of boolean queries: a prototype implementation and evaluation," *J. Am. Soc. Inf. Sci.*, vol. 44, no. 6, pp. 327–339, 1993.

[14] R. Agrawal, N. H. Gehant, and J. Srinivasan, "Odeview: The graphical interface to ode," in *Proceedings ACM SIGMOD Conference (1990)*, 1990.

[15] M. Angelaccio, T. Catarci, and G. Santucci, "Qbd: A graphical query language with recursion."

[16] I. F. Cruz, A. O. Mendelzon, and P. T. Wood, "A graphical query language supporting recursion," in *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. ACM Press, 1987, pp. 323–330.

[17] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," in *Proceedings of the 23rd International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1997, pp. 436–445.

[18] N. Murray, N. Paton, and C. Goble, "Kaleidoquery: a visual query language for object databases," in *Proceedings of the working conference on Advanced visual interfaces*. ACM Press, 1998, pp. 247–257.

[19] A. Sengupta and A. Dillon, "Query by templates: a generalized approach for visual query formulation for text dominated databases," in *Proceedings of the IEEE international forum on Research and technology advances in digital libraries*. IEEE Computer Society, 1997, pp. 36–47.