Query By Templates: A Generalized Approach for Visual Query Formulation for Text Dominated Databases*

Arijit Sengupta[†]

Computer Science Department
Indiana University
Bloomington, IN 47405
asengupt@indiana.edu

Andrew Dillon

School of Library & Information Science
Indiana University
Bloomington, IN 47405
adillon@indiana.edu

Abstract

The WWW has a great potential of evolving into a globally distributed digital document library. The primary use of such a library is to retrieve information quickly and easily. Because of the size of these libraries, simple keyword searches often result in too many matches. More complex searches involving boolean expressions are difficult to formulate and understand. This paper describes QBT (Query By Templates), a visual method for formulating queries for structured document databases modeled with SGML. Based on Zloof's QBE (Query By Example), this method incorporates the structure of the documents for composing powerful queries. The goal of this technique is to design an interface for querying structured documents without prior knowledge of the internal structure. This paper describes the rationale behind QBT, illustrates the query formulation principles using QBT, and describes results obtained from a usability analysis on a prototype implementation of QBT on the Web using the Java^{TM} programming language.

1 Introduction

The SGML Standard [11] has introduced a method for using documents as a means for information interchange and retrieval. We can now use properly structured documents not only for the traditional purposes of reading, browsing, and printing, but also for searching and querying. Automated searches in normal word-processor documents are usually restricted to linear word searches. However, users can use documents encoded in SGML to pose much more complex searches involving both text content and structure.

1.1 SGML, Databases and Documents

Traditionally, documents in electronic formats like text or word-processor documents are used mainly for word processing, editing, publishing, and possible reuse into other documents. The problem with these documents is that they can not be easily used for purposes other than what they are primarily designed for. The SGML standard extends the use of electronic documents beyond word processing and towards a means for information interchange. SGML was originally designed as a portable document encoding format for easy interchange between various platforms and systems. As stated in the standard [11], SGML was to be used "for publishing in its broadest definition, ranging from single medium conventional publishing to multimedia data base publishing." SGML achieves this goal by involving a modeling phase in document design and by completely separating layout and structure. The motivation in SGML is to define the structure of the document; and leave the layout to the processing application. The process for the design of document structure is conceptually similar to schema design in traditional databases. Applications processing an SGML document can then use this structural content as a database schema to solve queries (searches) on the document. This makes the treatment of documents encoded in SGML as databases for the purpose of querying a fairly straightforward task.

*Copyright 1997 IEEE. Published in the Proceedings of ADL'97, May 7-9, 1997 in Washington D.C. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyright and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: +Intl. 908-562-3966

†Partially supported by U.S. Dept. of Education award number P200A502367 and NSF Research and Infrastructure grant, award number NSF CDA-9303189.

1.2 Searching, querying and browsing

Traditionally, the most common method for retrieving information from documents is by perusing. For simple word-processor documents, perusing involves sequentially scanning or paging through the document to find the information of interest. This form of searching, which we technically call "browsing", depends completely upon the reader's perseverance and visual attention. The success of the search depends upon the reader's knowledge of the subject matter, the organization of the text, and similar factors. The recent advent of hypertext documents has changed the traditional sequential method of browsing. Browsing in hypertext documents is considered non-linear [14, 21] – usually performed by following links in the document to other related areas of the same document or entirely different documents.

Although hypertext documents may provide a more flexible way of browsing, finding information in hypertext documents by browsing still requires human intelligence, knowledge and experience. Whether readers are more comfortable with screen reading or paper reading is a debatable question[8]. Research shows strong evidence of paper as a more natural means of reading [7, 9] as well as a means to rectify the problems with reading from a screen [10, 16].

One major advantage of electronic documents over their paper counterpart is the capability of automated searching. Many information retrieval techniques have been proposed [20] that can quickly extract portions of documents matching given keywords. Similar searches can be quite difficult in paper documents[8]. However, keyword searches do not often retrieve the target documents. For large documents combining text with the logical structure of documents is often necessary. In many cases, many component search conditions need to be combined using boolean expressions to build a narrower search. For example, a reader might be looking for papers in a journal authored by "Jane Doe" with a title containing "creature". These types of searches are less common in traditional information retrieval systems which lack the capability of involving structure in searches. Searches involving structure and complex operations such as join have been the object of recent research [4, 1]. In this paper, we will refer to such searches as "queries".

2 Query languages for structured documents

The object of developing languages for the purpose of querying is to attain the capability of involving complex operations involving text (data) and structure (meta-data). Traditional query languages for relational databases have the capability of using the schema information effectively. Since the basic structure of relational databases consists of flat tables, query languages for relational databases do not usually deal with complex hierarchical structures. However, query languages for document databases need to effectively use the document hierarchy. Research on query languages for documents has unearthed a number of such languages. Two very useful surveys of such languages and systems can be obtained from [13, 3].

Most database systems implement specific database models, such as the relational model and Object-Oriented (OO) model. These database models are based on strong theoretical foundations and have formal procedural languages to manipulate data. In the case of relational databases, the most commonly used query languages are SQL(Structured Query Language) [2] and QBE (Query By Example)[22]. SQL is a textual language with a simple English-like syntax, and is widely implemented in most commercial database systems. QBE provides a visual method for posing queries and is suitable even for inexperienced users. This section describes some previous work done in the area of visual query formulation, and derives the motivation behind the current research.

2.1 Visual languages and interfaces

This paper presents the design of a visual interface for the purpose of querying document databases. Although many different types of procedural query languages have been proposed so far, not enough attention has been devoted towards providing visual methods for query formulation. Current implementations primarily use form-based approaches for building queries. In this section, we take a closer look at QBE and form-based querying.

Query By Example (QBE). Query By Example [22] is a visual language for querying relational databases. This language has a simple interface composed of tabular skeletons representing tables in the database. Users specify queries by entering sample values (or examples) in appropriate areas of the table skeleton. These values can be either search strings or variables for the purpose of join and other operations that require variables. QBE can also handle complex boolean combinations of such search expressions using a special section of the screen (termed *condition box*). Aggregation operations such as sum, count and average can also be performed by indicating appropriately in the tabular skeleton. Figure 1 shows a simple join operation using QBE as implemented in a popular database system. The main idea behind this method is that the user provides an example of outputs that she

expects from her query, and the query engine looks in the database for data that matches the given example. This works nicely for relational databases, primarily because the tabular structure of the database fits quite well with tabular skeletons used in the interface. Some useful properties of QBE are:

Simplicity The core QBE does not require the knowledge of any syntactic constructs or the internal structure of the database to use.

Equivalence relational databases are stored in tables, and in QBE, the queries are entered into equivalent table skeletons.

Closure Users give examples of the type of values that they are looking for, and the result of the queries shows up as tables in the same format as the data.

Completeness The core QBE, along with some additional commands, condition boxes and other constructs, can construct the same class of queries as relational algebra.

These properties make QBE a complete language that can adapt to any relational schema and make querying almost independent of the users' knowledge of the structure of the data. In a later section we will show how our approach keeps all these properties in a generalized interface designed primarily for documents, but applicable to any complex structured data.

Query By Forms. Although QBE is a formally accepted direct-manipulation visual language for relational algebra, most relational database systems still do not universally use or implement it. Some commonly used relational database systems implement different variations of QBE, but application developers designing query interfaces for a specific purpose seldom use the QBE method directly in their implementations. In these cases, developers use form-based interfaces.

In form-based interfaces, the user is presented with a list of searchable fields, each with an entry area that can be used to indicate the search string. To pose a query, the user needs to fill in the areas of the form relevant to her search. A more general form interface would have options of specifying boolean operations. As part of the comparison process in the current work, we implemented a version of the form interface that can work with current world-wide-web technology [see Figure 8]. Search interfaces employed by most web search engines also use interfaces similar to the one shown here.

2.2 Other query visualization approaches

Here we describe two experimental systems that use interesting visualization techniques for the purpose of information retrieval on complex structures. Although the approaches here are directed primarily towards visualization of data, they do provide useful incentive in this work.

TEXTVIZ. TEXTVIZ[6] is a system for intuitively visualizing, searching, and querying the contents of large text databases under development at TASC. This method uses a map-like representation of documents that shows users how components of documents are related to each other in the collection. It provides two levels of text database information – (1) a macro perspective, giving a top-level view of the whole database using the topographic map, and (2) a micro perspective focusing upon the conceptual content of each document component. TEXTVIZ uses a natural language or text processing front-end to extract meaning about the contents of a document database. It then generates and displays a text map portraying these contents. This provides the user the ability to comprehend the entire database at once, giving the user a global perspective of the organization of the contents of the database.

AIR and SCALIR. AIR (Adaptive Information Retrieval) and SCALIR (Symbolic and Connectionist Approach to Legal Information Retrieval) are two systems described by Rose et al.[18] that use a connectionist approach to the information retrieval problem. AIR's goal is to build a representation that will retrieve documents that are more likely to be relevant to particular queries. In order to achieve this, users begin a session with AIR by describing their information need using a very simple query language. Subsequently users can refine their queries by specifying multiple clauses or by navigating links shown by the system. SCALIR is an IR system for retrieving cases about copyright law. Internally, SCALIR contains a network of nodes representing terms, court decisions, and statute sections. A query consists of the activation of a few selected nodes. This activation is spread throughout the network, building the retrieval set from the activated nodes.

Although the methods described in this section primarily apply to data visualization, the direct manipulation of the data in its natural form and constant modification of the display based on the current status of the query has significant influence in the design of QBT.

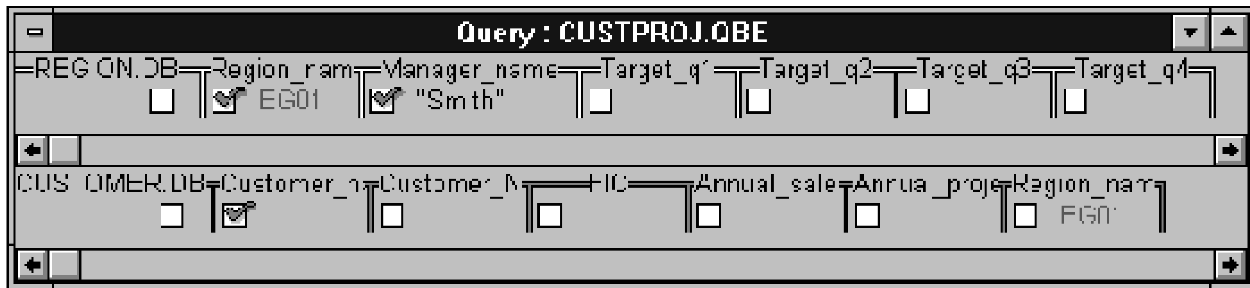

Figure 1: A sample QBE screen from Paradox for Windows

3 Query By Templates (QBT)

The current work generalizes QBE for databases containing complex structured data. QBE is very suitable for relational databases, since it uses tabular skeletons(analogous to tables in the relational model) as a means for constructing queries. In other words, the template for presenting queries in QBE is similar to the internal structure of the database. We use this idea to generalize QBE for any type of database in which each data instance has a simple visual template. In this generalized method (that we term "Query By Templates",) the basis of the interface is a visual template representing an instance of the database. Simple examples of templates are: a small poem for poetry databases, a table for relational databases, a sample word definition for a dictionary database, and a sample entry in a bibliography database. Any database that has a simple visual representation of its content can be used with QBT. For databases that do not have a general visual content, we can always revert to tables for the template.

The main goal for using QBT is to retain all the prominent properties of QBE. The intended properties of QBT that are analogous to those of QBE (as discussed in Section 2.1) are (i) simplicity, (ii) equivalence, (iii) closure and (iv) completeness. This section describes templates in greater detail to explain to basic design of QBT.

3.1 QBT: the basic design

At the simplest level, a QBT interface displays a template for a representative entry of the database. The user sees an example of the type of data she would expect to find in the database, such as a poem in a poetry database. She specifies a query by entering examples of what she is searching for in the appropriate areas of the template, and the system retrieves all the database entries that match the example she provided. To illustrate the interface, we use a simple template for a poetry database, as in Figure 2. In this figure,

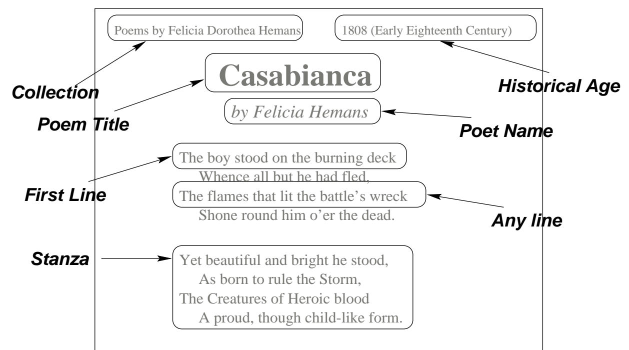

Figure 2: A simple template for poems, with its logical regions

we indicate a prominent logical region of the poem by circling it and labeling it with the corresponding region name. Physically the QBT interface consists of a small template image divided into areas corresponding to different logical regions in the database, as in Figure 2. Depending on the layout of the regions, the templates can be of different types, and these different types are described in the next sections.

3.2 Flat templates

As described in the previous section, QBT relies on the presence of a simple visual template for the instances in the database. In most cases, this template could be planar or flat. This means that all logical regions of the template can be displayed simultaneously in a two dimensional image without overlapping(like in Figure 2). We call these templates "flat templates". Flat templates are usually easier to display and use,as the structural regions can be simultaneously displayed in a plane, possibly by showing multiple instances of some regions. For example, in Figure 2, the *First Line* and *Any Line* regions are subregions in *Stanza*. To display these subregions, the template needs to include a second stanza which is broken into its components.

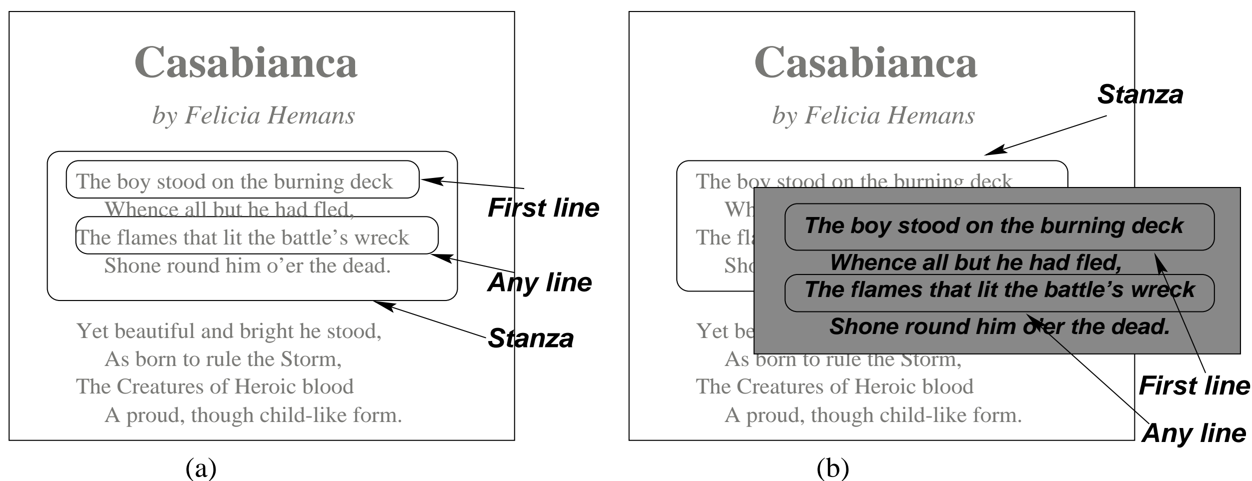

Figure 3: Templates with (a) Embedded Regions and (b)Recursive regions

3.3 Nested templates

Although flat templates are easy to display and navigate, they cannot model structures with deep levels of nesting. In this case, we use templates that can be nested. In nested templates, regions are allowed to overlap. In particular, certain regions can be completely inside other regions to represent subregions. To display embedded logical regions, we use one of the following methods:

Embedded Regions In this method, subregions are displayed inside the parent region. As in flat templates, all regions are displayed simultaneously in the same plane of the image. Component regions no longer need to be mutually exclusive. This method is a simple extension of flat templates, but it makes templates much more powerful while retaining the simplicity. However, this method is again limited to structures in which the nesting level is not very deep and the top-level region is physically large enough to include all the nested regions without completely obscuring itself. An example of this type of nesting in shown in Figure 3(a).

Recursive regions This is the most general method of nesting regions. In this method, a region with subregions can be recursively expanded. During traversal, the user may "zoom in" to a parent region to display its subregions. The magnified portion of the template can be an independent template, and can be subsequently magnified to get to additional levels of nesting. Although this method can capture any general structure, the templates have to be cleverly designed so that users are not disoriented by the nested templates. Figure 3(b) shows this method of displaying internal structures for the same poem example.

3.4 Structure templates

Structures, particularly large ones, may get too complex to use nested templates. In these cases, it is often necessary to display the internal structure simultaneously with a template that displays the relative position of the current region. As mentioned earlier, most documents can be thought of as having a hierarchical structure that can be conveniently visualized as a tree. Showing a template simultaneously with a hierarchy of logical regions depicting the context simplifies the nested structure visualization. An example of the structure template is shown in Figure 7, which is a screen-shot from the prototype implementation of QBT, described in Section 5.

4 Querying with QBT

Normal keyword searches bounded by structural regions are simple in the QBT interface. As described earlier, users express their queries by indicating the search keywords in the appropriate regions of the template. In this section, we show all the different types of possible searches that can be performed with QBT.

One can treat QBE[22] as a special case of QBT where the templates used are table skeletons that instantiate tables in the database. In QBE, queries are specified by entering values in proper positions of the tables. These values may be either constants (i.e., strings or numbers), variables (or *examples*, usually differentiated from the constants by underlining), or expressions involving constants and variables combined with arithmetic and comparison operators. The output of the query is specified by marking the regions that need to be presented in the output. QBT uses the

Figure 4: Query formulation with QBT: (a) Simple selections and (b)Logically combined selections

same basic principle, with the extension that the templates are not restricted to table skeletons but can be any visual representation of the database instances.

The primary difference between the method of expressing queries in QBT and that in QBE lies in the fact that the templates in QBE are essentially one-dimensional. Although QBE uses a 2D tables for querying, the meta-data (attributes of the relations) only appear along the horizontal axis as column headings of the tables. QBE uses the rows to specify multiple search conditions and logical operations between the search conditions (see examples in [22].) In QBT, the regions (meta-data) are distributed along both dimensions of the template, utilizing whole template plane for visualizing the structure. Logical operations between regions can be expressed by physically connecting two or more regions via a logical operator. Logical operations within regions can be formed using logical expressions within the scope of that region. In the rest of this section, we discuss how different types of queries are performed using QBT.

4.1 Simple selections

Simple selections include searching for constant strings or numbers within logical regions of the document (the whole document itself being one region). In QBT, such searches are performed by simply entering the search string in the corresponding region of the template. As a result of such a search, database instances rooted at a default region that match all the specified conditions are returned. In other words, the given search criteria are combined using a logical conjunction operation. The result of the query is by default rooted at a preselected region defined by the template. However, users can mark the regions that they want returned by placing a print-marker on them.

In the illustrations (see Figure 4), the small tick-mark ($\surd$) is used as a print indicator. In the examples, Figure 4(a) denotes the simple query: "Find the poem titles and poets of all the poems that have the word hate in the title and the word love in the first line." Note that unlike QBE, searches are substring matches instead of exact string matches. So, entering the word "love" in the region "first line" matches all poems with the first line containing the word "love" anywhere in the first line. In QBE, this is done by indicating examples before and after the search string. In case of documents, substring matches make more sense since exact searches are far less common.

4.2 Selections with multiple conditions

We have just seen that if multiple conditions are specified in different regions, they are combined using logical conjunctions, implying that the results returned from the query will satisfy all the specified search conditions. If this is not desired, search conditions can be combined using logical operators *AND, OR, NOT*. Negation of individual conditions are done by placing the keyword "NOT" in front of the string. Implementations of the interface may use some visual mechanisms to place this negation keyword. Connecting various search strings using the binary operators "AND" and "OR" involves simply connecting the two strings using a pointing device and selecting the proper operation type for that connection. Figure 4(b) demonstrates how this is done by expressing the query: "Find the poem titles and poets of all poems that either do not have the word 'hate' in the title or have the word 'love' in the first line." Notice the introduction of the negation and the "OR" connection.

Note that if multiple clauses are connected using logical constructs, the order in which the expressions are evaluated depends on the direction of the arrow. However, it is possible to override this order of evaluation by placing parentheses in appropriate places in a condition box, which we will demonstrate shortly (see Section 4.4).

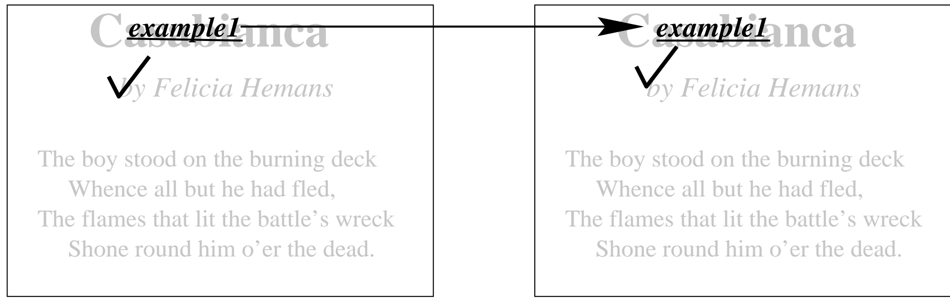

Figure 5: Query formulation with QBT: Joins

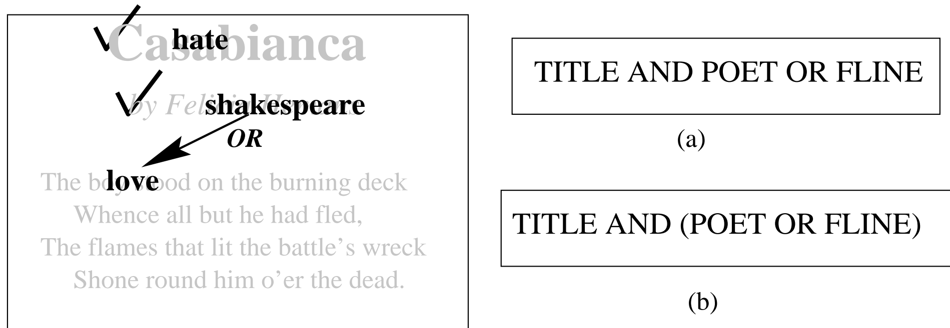

Figure 6: Changing precedence of operations with Condition boxes

4.3 Joins and multiple templates

In this section, we look at a special class of query called "join". A join is an operation using which multiple fragments of a database are combined together based on some common property. Joins are indispensable in relational database, since the relational design involves "normalizing" a schema by breaking it into flat tabular fragments. This fragmentation necessitates combining the individual fragments together at the time of query processing using the join operation. However, in document databases, the structure is not normalized into planar fragments, but allowed to grow hierarchically, so joins are not required to combine fragments. However, joins are still useful to solve queries that requires comparison of different parts of a database or different instantiations of the same database.

For example, one may try to "find the pairs of poets who have at least one poem with a common title" (as in Figure 5). In this case, we need to generate two instances of the poetry database and run the query comparing the titles of the two poems. This is achieved in QBT by using multiple templates. In the case of the above query, the same template is instantiated twice, and the join attributes are connected together. The connection can be augmented with comparison operators to specify joins other than "equi-joins" (joins that use equality on the join attributes). Once again, in the case of asymmetric comparison operations, the precedence of the operators is determined by the direction of the arrow. Although we are still using underlined examples to highlight the joins, they are redeemed unnecessary by the use of the joining arrow.

4.4 Queries with complex conditions

Like QBE, QBT cannot always enable visualization of complex conditions involving more than two regions. These conditions need to specified using a condition box. The condition box can also be used to override the precedence of operators. As search strings and examples are specified, the condition box is automatically updated. The user can then insert parentheses as necessary to change the default precedence. For example, in Figure 6, if the default precedence is used, the query evaluates to: "Find the poem titles and poets of the poems in which either the title contains 'hate' and the poet is Shakespeare, or the first line contains 'love'." The default condition box is shown in Figure 6(a). However, this default can be changed to: "Find the poem titles and poets of the poems in which the title contains 'hate', and either the poet is Shakespeare, or the first line contains 'love'." The condition box of this query is shown in Figure 6(b).

The condition box can also be used for specifying more complex conditions involving more than two variables in an expression. The condition box functions in the same manner as the condition box in QBE.

5 A prototype implementation of QBT

We built a prototype[1] of the QBT interface using the Java^{TM} programming language. The prototype implements most of the features described here including the embedded template (without recursive magnification) and the structure template. We have not yet incorporated the condition box in this prototype, but it will be added in the next release. We also included an experimental version of an SQL language translator from the QBT query. Figure 7 shows two parts of the screen – one showing the template screen and the other showing the structure template.

As an experiment, we used the Chadwyck-Healey English Poetry database with templates similar to those described in this paper for performing the queries. We used the Java^{TM} language to build the user interface. Queries generated using the interface are sent to a query engine by the HTTP (HyperText Transfer Protocol) protocol, which is run from a web server as a CGI (Common Gateway Interface) executable. The engine generates its output in HTML which is displayed by the clients. We wrote the engine in C/C++, using the API (Application Programming Interface) provided by the Pat [17] software. However, due to limitations of the processing capabilities of Pat, this engine does not support variables or joins. A part of the current research is aimed at building a database engine that will support queries such as join, grouping, and nesting, without converting the documents into a different database format.

6 Usability analysis

We performed an initial phase of usability analysis on the prototype interface. The main goal of this analysis was to detect differences between this interface and a standard forms-based interface with similar search capabilities. In particular, we were interested in differences with respect to (1) accuracy, (2) efficiency and (3) satisfaction. This section describes the method used during the experimental evaluation of the new Java-based (QBT) interface.

6.1 Experimental design

The experiment consisted of two primary parts. In the first part, we gave the subjects ten questions – among which we prepared nine, and left the tenth question open to the subject's imagination. All subjects were given the same set of questions (see Appendix A.) The questions varied in complexity and were designed so that all except one returned some matches. The subjects were to answer the questions using the search interface and to write down the number of matches returned by the search, after making sure that the question was interpreted properly by the searching program. The independent and dependent variables for the experiment are outlined below:

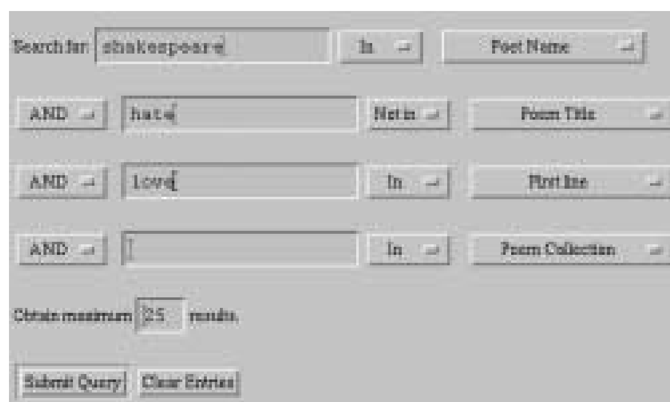

Figure 8: The form implementation of the query interface used in the usability analysis

Independent variables:

A. *Interface type*, (1) New Java-based QBT interface, and (2) generic form-based interface;

B. *Subject type*, (1) expert, and (2) novice.

Dependent variables:

1. *Efficiency*: The amount of time in seconds the subjects take to answer each question.
2. *Accuracy*: The degree of accuracy of the answers.
3. *Satisfaction*: How satisfied the users were after using the interface (measured by self-reports in written debriefings.)

Twenty subjects were chosen for the experiment. The experiment was conducted using a "between-users" strategy[19], where two distinct groups of users use the two platforms. In our experiment, ten subjects were given the Java-based interface (see Figure 7), while another ten (five novices, five experts) were given the form-based interface (see Figure 8).

After the subjects finished the questions, we asked them to complete a small survey of their experience with databases and query engines. The survey also asked them to compare the interface with a standard web-search interface that they have used.

[1] The current version of the prototype can be seen at: http://blesmol.cs.indiana.edu:7890/projects/SGMLQuery. Note that only the interface is accessible from outside Indiana University. The results of the queries cannot be viewed from a remote location because of copyright restrictions.

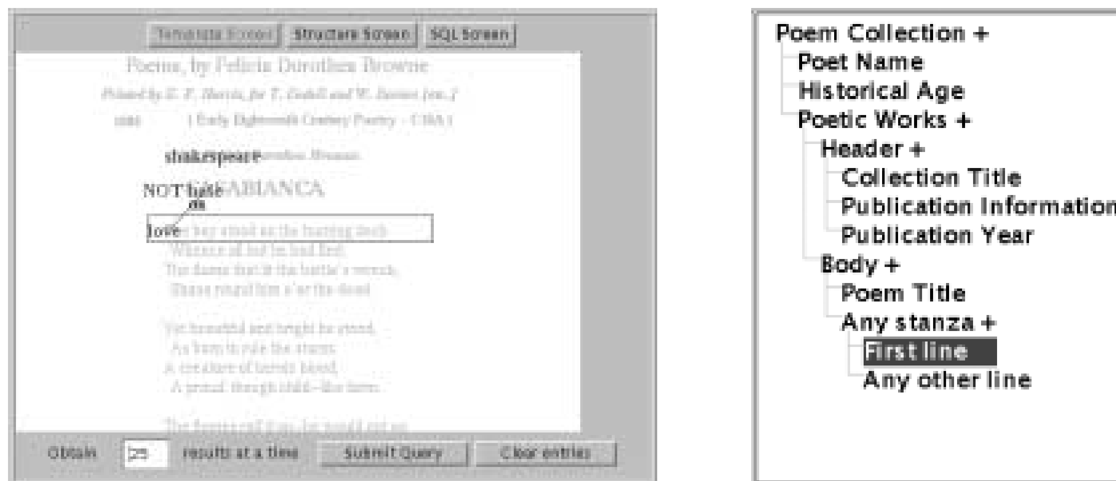

Figure 7: A screen shot-from the prototype showing the template and structure

6.2 Subjects

Subjects were students who volunteered to participate in the research. There were no major requirements from the subjects except that they all be Indiana University affiliates, because of the copyrighted nature of the database on which we performed the experiment. We divided the subjects into two groups based on their experience with computers and databases. The subjects classified as "novices" had minimal computer expertise – generally limited to only e-mail and occasional World Wide Web access. The subjects classified as "experts" were people accustomed to using databases and the web as well as designing and programming graphical user interfaces. We made no distinctions between male and female subjects, since it was not one of the independent variables in this analysis (eleven female and nine male subjects participated in this study.)

6.3 Equipment - software and hardware

We performed all the experiments using Netscape 2.0 for the Java-based interface and either Netscape 2.0 or 1.1 for the form-based version. For the Java-based interface, we restricted experiments to machines having 16MB or more system memory, since Netscape's Java performance is sub-standard with less memory. However, for the other interface, no memory restriction was enforced since the HTML forms do not have additional memory requirements.

6.4 Data collection

We collected two types of written data: (1) the subjects' responses to the survey questions and (2) the subjects' responses to the number of matches for each search problem (please see Appendix A for the actual questions). The subjects were timed automatically by the server and the query engine that was actually executing the queries.

Basic procedure. The subjects were introduced to the experiment and the target interface. After an initial introduction, the subjects were given the experimental queries and asked to input them sequentially, and to note down the number of matches returned by the database for every query. They were also asked to verify their results to eliminate typographical errors by checking the response from the database and looking at sample results from their search. After they finished the queries, they were given a set of survey questions which they were to answer. They were also asked to verbally describe their feelings and general reactions from their use of the systems.

Experimental queries. A set of nine queries (see Appendix A) were given to each subject. For the tenth query, they were asked to search for something of their own interest. The first and the easiest query was primarily meant for the subjects to get acquainted with the system, and the last query was mainly to see what types of questions users often search for. The queries ranged from very simple searches involving a single clause in a field to complex searches involving up to four clauses combined together.

Timing technique. The subjects were timed by electronic means. Whenever a user submitted a query using either interface, the server logged the access time. Also, the query engine that we designed logged timing and other detailed information about the queries sent by the users. We also designed the Java interface so that it would send log messages to the server. This al-

lowed the server to keep track of all the actions (such as button press and query selection) that the user took over the course of submitting the queries.

Survey questions. In addition to the queries, we gave the users a small set of questions to express their experience, preferences, and the degree of satisfaction with the interface. They were also asked to point out features that they liked or disliked in the system that they used.

General feedback. After the experiment was over, the subjects were asked to comment on their general feelings about the project; and their comments and suggestions were noted. This data was primarily used for the purpose of designing improved features for the current interface.

7 Results

This section describes in detail the results that we obtained from the usability analysis. We divide the results in three different sections, one each for the dependent variables — *accuracy*, *efficiency*, and *satisfaction*. For each of the measures, we performed a multivariate ANOVA test with a .05 significance level. In addition, for the accuracy and efficiency measures, we show the results for all ten questions, and make appropriate inferences based on the results.

In the following analysis, for the independent variable "Interface type", the Java interface (Figure 7) is given a value of 1 and the form interface (Figure 8) is given a value of 2. For the independent variable "Subject type", the values of 1 and 2 are assigned to experienced and novice users, respectively. The tasks are denoted as "qns1" through "qns10".

7.1 Accuracy

We took the accuracy measures by evaluating the answers to each question in $0-5$ scale. Perfect answers were given 5 points and completely wrong answers (of course, there were none in the experiment) were given 0 points. Based on the type of mistake the users committed, their answers were given a value between 1 and 4. For the other questions, there was no significant difference between either the two interfaces or the two groups of users. The multivariate ANOVA analysis did not show any significant effect of either the interface type or the user type, or a combination of both with respect to accuracy. See Table 1 for a summary of the significance values for the ten questions.[2]

[2]For questions 1,2,4 and 10, all the participants got the correct result, so there was no variance in the outcome of these questions.

7.2 Efficiency

For the efficiency measure, we used the time (in seconds) between two successive submissions of queries. The absolute times at which (1) the system was first accessed and (2) the queries were submitted, were logged by the query processing system, and we calculated the difference between these times to get the time taken for each task by the subjects. For the first task, we used the time difference between the first access of the search page and the submission of the first task. This turned out to be a problem (as indicated by the results,) since the Java interface page did not have any other text besides the search interface itself, while the form interface had some instructions in it which most of the subjects spent time reading, before composing the first query. Table 2 shows that experts were significantly more efficient than the novices on both the types of interfaces. However, except for Task 1 and Task 7, there was no significant efficiency difference between the two interfaces.

For Task 1, the subjects using the Java interface performed significantly better than the subjects using the form interface, and the possible explanation is given above. For Task 7, the users of the form interface performed significantly better than the users of the Java interface. On a detailed analysis of the subjects' actions, we found that this task required the users to switch to a different screen for the Java interface, and most of the users could not understand the necessity for this action. This will be rectified when all three screens are displayed simultaneously; as users will not have to discover something new in order to perform this query.

Table 2 displays the results we obtained from the multivariate tests of significance. The values show clearly that the combined effect of both the interface and subject type was significant only for Task 1. The effect of subject type was definitely significant on almost all the tasks - the experts were significantly more efficient on both the interfaces. However, the interface difference did not have any significant effect on all the tasks except Task 1 and 7; and the reasons are already explained above.

7.3 Satisfaction

For the satisfaction measure, the users were asked to grade the interface that they used in a scale of five qualitative values: *Much better, Little better, About the same, Worse, Absolutely worse*. These five classes were given the ranks 5,4,3,2 and 1 respectively. This data was taken after all the actual tasks were performed, and was not calculated on a task-by-task basis. As before, a Multivariate ANOVA measure was

Effect\Trial no.	Qns1	Qns2	Qns3	Qns4	Qns5	Qns6	Qns7	Qns8	Qns9	Qns10
Subject	-	-	0.332	-	0.624	0.332	0.743	0.332	0.811	-
Interface	-	-	0.332	-	0.153	0.332	0.332	0.332	0.477	-
Interaction: subject & interface	-	-	0.332	-	0.624	0.332	0.201	0.332	0.477	-

Table 1: Significance values for Accuracy

Effect\Trial No.	Qns1	Qns2	Qns3	Qns4	Qns5	Qns6	Qns7	Qns8	Qns9	Qns10
Subject	0.003	0.085	0.029	0.220	0.000	0.004	0.001	0.000	0.018	0.004
Interface	0.000	0.726	0.696	0.522	0.420	0.660	0.001	0.452	0.897	0.927
Interaction: subject & interface	0.006	0.379	0.446	0.494	0.830	0.054	0.066	0.101	0.934	0.912

Table 2: Significance Values for Efficiency

taken on this data. This clearly shows a significant effect of interface on the satisfaction of the subjects. However, the satisfaction was independent of subject type or the combined effect of interface and subject types (see Table 3).

8 Discussion and conclusion

QBE and forms are both quite popular means for querying in the relational domain. The main advantage of the form interface is that it is very simple to implement and easy to use for small databases. However, forms do not adapt very well for databases with a very complex structure, and most text-based databases tend to have a very complicated structure (for instance, the Chadwyck-Healey database used in the prototype contains over fifty logical regions.) A form interface that can search on only a few of these areas is easy to construct, but if the number of searchable regions is increased, the usability gets questionable. On the other hand, a structure template can be easily used to navigate through complex hierarchical structures. Even for complex hierarchies, the focus can be concentrated in the regions of interest using advanced methods like differential magnification [12].

Another advantage of the template method is its direct relationship to the internal structure of the database. Forms always look the same, whether the underlying database is a poem, a dictionary, a quotation collection, or even a relational database. However, templates can be custom-designed for different types of databases. This way, templates can provide a direct reflection of the users' mental models [5, Ch.6], a significant factor in the design of good user-interfaces. Moreover, templates use the principle of familiarity [15], which is demonstrated to work well for novice users. The only disadvantage of templates is that good templates require expensive graphics terminals, while forms work quite well with terminals without graphics capabilities. However, with the advance in technology, non-graphics terminals are less common, so the assumption of a graphics-capable terminal is not very demanding.

The implementation of QBT in this work is in an early developmental stage, and has substantial potential for improvement. The experiment we performed clearly indicated some of the ways it could be improved. However, in spite of being a prototype interface, this QBT implementation demonstrates that QBT is suitable for querying textual databases using a simple graphical interface. Moreover, QBT is at least as efficient and accurate as the general form-based approach, and significantly more satisfying to the users. We believe that the idea behind QBT will give us a starting point for query interfaces in future text database systems. A significant portion of the current research is aimed towards the theoretical stability and soundness of the QBT concept, and once established, this method has the potential of being the standard querying mechanism for text databases.

Acknowledgments

We are very thankful to the subjects of our usability analysis for their participation and cooperation. We are also grateful to the LETRS (Library Electronic Text Resource Service) subdivision of Indiana University Library, and especially ex-directors Richard Ellis and Mark Day for allowing us to use the Chadwyck-Healey Database for this research. We would also like to thank Prof. Dirk Van Gucht, Computer Science, Indiana University, for his careful reading and useful comments.

A Tasks used in the usability testing

1. Find the poems written by “Shakespeare”.
2. How many poems were written in the Middle English Period age (MEP)?
3. Find all the poems written in the Early 19th Century period (C19A) that have the word “burning” in the first line.
4. Find the poems that have the word “hate” in the title and the word “love” in the first line.

Effect	Interface and Subject Type	Subject Type Only	Interface Only
Significance values	1.000	0.503	0.014

Table 3: Significance Values for Satisfaction

5. Find the poems not written by "Hemans" that have the word "wreck" somewhere in a stanza.
6. Find the poems written during the Early 18th Century (C18A) which have the word "love" in the collection title, as well as in the poem title, but not in the first line.
7. Find the poems that have the phrase "expostulation and reply" anywhere in the body of the poem.
8. Find the poems written by Keats that do not have the word "mortal" in any of the stanzas.
9. Find the poems written by Shakespeare that has the phrase "to be or not to be" somewhere in the poem body.
10. Write a query of your own from your interest in poems, and indicate the number of matches you found for that query.

References

[1] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. *Proceedings, 19th Intl. Conference on Very Large Data Bases*, pages 73–84, 1993.

[2] American National Standards Institute, New York. *ANSI X3.135-1992 and ISO/IEC 9075:1992, Database Language SQL*, 1992.

[3] Ricardo Baeza-Yates and Gonzalo Navarro. Integrating contents and structures in text retrieval. *SIGMOD Record*, 25(4):67–79, March 1996.

[4] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Efficient text searching of regular expressions. *Proceedings, 16th International Colloquium on Automata, Languages, and Programming*, pages 46–62, 1989.

[5] Paul Booth. *An Introduction to Human-computer Interaction*. Laurence ErlBaum Associates Publishers, 1989.

[6] Nathan Combs. Large text database visualization. *Advances in Classification Research, Proceedings of the 3rd ASIS SIG/CR Classification Research Workshop*, 3, Oct 1992.

[7] A. Creed, I. Dennis, and S. Newstead. Proof-reading on VDUs. *Behavior and Information Technology*, 6(1):3–13, 1987.

[8] Andrew Dillon. *Designing Usable Electronic Text: Ergonomic Aspects of Human Information Usage*. Taylor & Francis, London; Bristol, PA, 1994.

[9] J.D. Gould, L. Alfaro, V. Barnes, R. Finn, N. Grischkowsky, and A. Minuto. Reading is slower from crt displays than from paper: Attempts to isolate a single variable explanation. *Human Factors*, 29(3):269–299, 1987.

[10] J.D. Gould, L. Alfaro, R. Finn, B. Haupt, and A. Minuto. Reading from crt displays can be as fast as reading from paper. *Human Factors*, 29(5):497–517, 1987.

[11] International Organization for Standardization, Geneva, Switzerland. *ISO 8879: Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, 1986.

[12] T. Alan Keahey and Edward L. Robertson. Techniques for non-linear magnification transformations. In *Proceedings, Visualisation '96 Information Visualization Symposium*. IEEE, October 1996.

[13] Arjan Loeffen. Text databases: A survey of text models and systems. *SIGMOD RECORD*, March 1994.

[14] Ted Nelson. *Literary Machines*, volume Version 87.1. Published by author, 1987.

[15] Donald Norman. *The Design of Everyday things*. Doubleday Currency, 1990.

[16] D. Oborne and D. Holton. Reading from screen versas paper: there is no difference. *International Journal of Man-Machine Studies*, 28(1):1–9, 1988.

[17] Open Text Corporation, Waterloo, Ontario, Canada. *Open Text 5.0*, 1994.

[18] Daniel E. Rose and Richard K. Belew. Toward a direct-manipulation interface for conceptual information retrieval systems. *Interfaces for Information Retrieval and Online Systems - the state of the art*, 1991.

[19] Jeffrey Rubin. *Handbook of Usability Testing: How to plan, design and conduct effective tests*. John Wiley & Sons, Inc., 1994.

[20] Gerard Salton. Developments in automatic text retrieval. *Science*, 253:974–980, 1991.

[21] Ben Shneiderman and Greg Kearsley. *Hypertext Hands-on! An Introduction to a New Way of Organizing and accessing Information*. Addison-Wesley, 1989.

[22] M. M. Zloof. Query by example: A database language. *IBM Systems Journal*, 16(4), 1977.