Proceedings of the 17th Association of Computing Machinery Scientific Applications Conference. Madrid, Spain. March 10-14, 2002.

Iterative Querying in Web-Based Database Applications

Ramzi A. Haraty and Mazen Hamdoun
Lebanese American University
P.O. Box 13-5053 Chouran
Beirut, Lebanon 1102 2801
Email: rharaty@lau.edu.lb

Abstract

Web applications are increasingly relying on databases. The traditional method offered by most web sites of submitting a query across the Internet and getting a response from a server packed in a Hyper Text Markup Language page is no longer satisfying [1][3][4][5]. This is due to the result set: it is often too large or empty. This forces the user to try and guess which constraints should be relaxed or tightened in order to obtain the desired result set. And so, the user goes through several submit/response cycles. Web applications that require this sort of interactive exploration of databases need to use a model that is different from the submit/respond model described above. In this work, we propose an iterative querying model that integrates querying with result browsing.

Keywords: Browsing, iterative querying, result set, and web-based database applications.

1. Introduction

In traditional databases, queries are rigid in that they are intended for asking very specific questions [2]. The results are interesting no matter what the result set is. The individual query itself is the goal. On the other hand, when we are interested in exploration the goal is locating particular records of interest. To do this, we need to provide the user with a combination of result browsing and searching so that he/she can see the query and its results simultaneously. As the user changes the query constraints, the impact on the result set should be immediate.

The aim of this work is to provide how the overall design of such a system might be with some stress on the user interface, which is a key element in the exploration process. The rest of the paper is organized as follows: section 2 presents the user interface. Section 3 details the overall system design. Section 4 concludes the paper.

SAC 2002, Madrid, Spain
© 2002 ACM 1-58113-445-2/02/03. . .$5.00

2. User Interface

The user interface plays a key role in enhancing the experience of the user. Records are displayed in a scrollable list format with a separate column for each numeric and categorical attribute as shown in Figure 1. At the top of each column is a title bar showing the name of the attribute. The names of the attributes are obtained from the database catalog. Beneath each of these titles is an "attribute control" used for specifying attribute restrictions. Categorical attributes are represented by select lists that allow users to (de) select (un) desired values. Numeric attributes are represented by vertical sliders, which may be resized and dragged to specify desired ranges. Initially, these attribute controls are maximized to include the full value-range of each attribute. Immediately, a user can see how many records are available, the value range of each attribute, and actual matching records. At any stage, the user can scroll through the list of records in the result set and can instantaneously change the sort order of the record list by clicking on the title bar of any column.

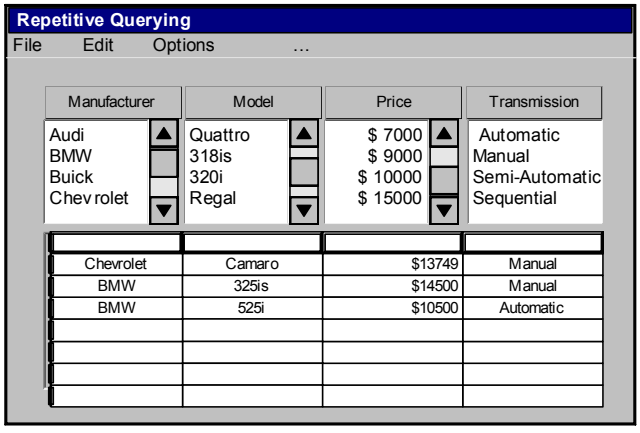

Figure 1. The Graphical User Interface.

There is no "submit" button. As the user adjusts the attribute controls under each column's heading, the records in the view pane are continuously updated to reflect new restrictions. The record count is also continuously updated during these adjustments. This lets the user know immediately if the query is becoming too restrictive and whether or not the user should continue adding specifications.

Contrast this iterative querying model with the conventional form-based approach [3]. The user would be entering

requirements into a static query page and would not know the result of the actions until the query is submitted and the response is received. If the query was restrictive, the user would most likely be presented with a short message indicating that no records matched the criteria, and that the user should try again. If the query was too loose, the user would probably be shown a single large listing of all matching records, or more likely, a series of linked pages containing twenty or so matching records per page. In both cases, no information is given as to how the query is to be relaxed or tightened, forcing the user to repeat this submit/response cycle as the user adjusts the requirements unaided. Each such cycle requires the attention of the web and database servers, not to forget a round-trip through the network. In the case of iterative querying, empty result sets can be easily avoided or backed out because the user can actually see the result set change as the query is tightened or relaxed.

2.1 Exploration Based on Example Records

Since query results are always visible, this allows a Query-By-Example [6] capability not available in most search interfaces. Rather than explicitly specifying a desired range of attribute values, a user can select example records to direct the exploration. This type of querying can be very useful in situations where the user may not be too knowledgeable about the domain, but knows of some sample records of interest. For example, a user who does not know about horsepower or what a "fast" 0-100-kph time is can still search for sporty cars by selecting several known examples. These records implicitly define a query region comprising the smallest hypercube that contains all the selected data points.

2.2 Exploration Based on Similarity

In addition to querying by example, we can allow sorting by example as well. This is essentially similarity search, except that instead of treating similarity as a separate stand-alone query like some search engines do for web pages (e.g., Excite) [7], it is integrated into the existing view. After selecting one or more sample records, the user can click a button and sort all the records by how "near" they are to the selected examples. Nearness is based on some similarity model. The similarity scores are then displayed in a new column labeled "Rank" which appears after the rank button is pressed.

3. System Design

We now describe the details of how we implement iterative querying. In order to obtain interactive response times, we exploit several key observations. First we must cache data records in the local client. There is little chance of interactive response times unless the interaction between the user and data is moved off of the server and out of the network. While this does place a memory requirement on the client, it offers an advantage beyond that of just faster access - the cache can be tailored to the user and the particular task of data exploration. An initial query representing the set of data to be explored is used to transfer data from the server-side database. This data is then cached using special data structures for further exploration. In cases where users may wish to explore data outside the current cache, techniques such as semantic data caching can be used [8].

Fortunately, while the total size of the underlying database can be huge, the scope of the data over which a user performs interactive exploration for some task is often limited enough that corresponding records are able to fit in a reasonably sized cache. For example, in web-based product exploration applications, a user will explore options within one product category at a time and not across product categories. The active set thus typically consists of records in thousands and not millions.

We also take advantage of the fact that there is always a notion of a current state. This state is represented by the current settings of the attribute controls (i.e., the "query") and the corresponding set of matching records. Every adjustment of these controls represents a minor change to an existing query. Rather than execute the new query entirely from scratch, we only need to update the current results to reflect the change. This can be made extremely efficient because of another observation: there is only one mouse. The user can adjust the restrictions of only one attribute at any given instance.

By taking advantage of these observations, we are able to exhibit extremely fast response times, even with datasets containing hundreds of attributes and hundreds of thousands of records.

3.1 Architecture

Figure 2 gives a high-level overview of the proposed architecture. The *DataColumn* objects represent the data cache with each *DataColumn* representing one column of attribute data. *DataGroup* maintains this cache. Observe that with this design, data for different attributes can be loaded and processed asynchronously. Thus, during data loading, the user can see column data appear progressively, rather than having to wait for the entire dataset. Furthermore, once a column of input data has been loaded and its *DataColumn* object created, the user may immediately begin to restrict or sort on that column, even as other columns are still being loaded.

ListRenderer represents the Graphical User Interface (GUI). It is responsible for rendering the current set of matching records, as well as passing user changes in attribute restrictions to the core engine. This interaction is handled by events and explicit Application Programming Interface (API) calls.

The final piece labeled *Core* encompasses the entire design. In a Java implementation, this is the hosting applet that must deal with menus, graphical layout and other details. It is also responsible for determining what data source to use and what attributes to select. This is handled through user interaction. The *Core* component is also responsible for retrieving the data from the data source. It can do so through a standard API we call the *DataPump*. This design allows us to support various data sources by simply plugging in a different implementation of the *DataPump* API. In the particular instance shown in Figure 2, the

DataPump communicates via HTTP to a Java *Servlet* running on a web server. This *Servlet* uses JDBC to communicate with the database. When *Core* requests data via the *DataPump* API, the *DataPump* object passes the request to the *Servlet*, which in turn issues a query to the database. Data is returned to *Core* from the *DataPump* in column-order rather than row-order. This allows each column to be converted into *DataColumn* objects independently and asynchronously. How data is transferred between the server and the *DataPump* (row-order vs. column-order, synchronous vs. asynchronous) is entirely up to the *DataPump* implementation.

For concreteness, we will be describing the design using the example dataset shown in Figure 3. This dataset represents car listings and contains attribute data for the ***Make*** of each car and the ***Distance*** in miles between the user and seller locations.

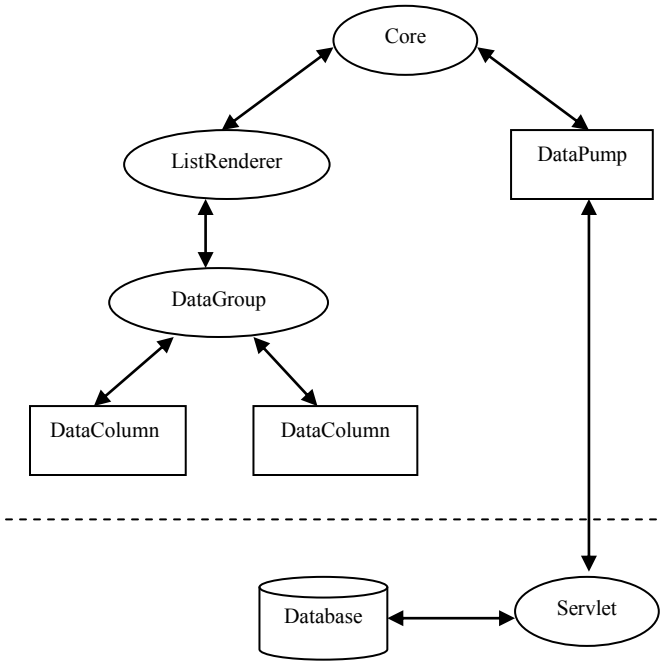

Figure 2. Architecture of the Model.

Note that this example demonstrates how the data received and cached by the system can be tailored for a particular user - ***Distance*** is a derived attribute that depends upon the user and would not exist in the server's database. Each record in this dataset is uniquely identified by an integer in the range [*0, n-1*], called a record identifier, or RID. These RIDs are not part of the dataset and do not correspond to any database or server-side identifiers. They are simply internal IDs used in the cache, assigned sequentially as records are loaded from the server.

RID	***Make***	***Distance***
0	*Ford*	*23*
1	*Honda*	*3*
2	*Ford*	*537*
3	*Chrysler*	*117*
4	*Honda*	*64*
5	*Honda*	*8*
6	*Chrysler*	*363*
7	*Ford*	*192*
8	*BMW*	*41*
9	*Honda*	*89*
10	*BMW*	*207*

Figure 3. A Dataset Example.

3.1.1 DataColumns

Each *DataColumn* is responsible for caching and indexing one attribute's worth of data. *DataColumn*s consist of two basic components: an array of data values indexed by record identifier, and a *RID-list*. The *RID-list* contains RIDs sorted by their corresponding attribute value. *RID-list*s are required for performing fast attribute restrictions, but offer the additional benefit of pre-computed sorts. The *DataColumn* objects differ for numeric and categorical data types.

Numeric DataColumns

For numeric attributes, we allocate an integer or float array and copy incoming data values into it. At the same time, we allocate an integer array for the *RID-list* and store the RID values in order (from *0* to *n-1*, where *n* is the total number of records). The *RID-list* is then sorted by the corresponding attribute values, resulting in a final *DataColumn* object like the one shown in Figure 4. Observe that *Data [RID-list [i]]* gives the value of the i^{th} item in the sorted list of data values.

Categorical DataColumns

Categorical *DataColumn*s are somewhat more involved than numeric ones. In addition to a data array and a *RID-list*, categorical *DataColumn*s also require a hash table and specialized objects called *RIDSet*s. This is due to the fact that users need to be able to add and drop arbitrary categorical values to and from the query. A *RIDSet* object consists of three components: a categorical string value, an integer count value, and an integer index.

RID-list	**Data**
1	*23*
5	*3*
0	*537*
8	*117*
4	*64*
9	*8*
3	*363*
7	*192*
10	*41*
6	*89*
2	*207*

Figure 4. *RID-list* and Data.

We first describe the creation of the categorical data array. Instead of allocating an array of strings to store the data values, an array of *RIDSet* references is allocated. Also allocated is an empty hash table. As the incoming categorical attribute data is

scanned, the hash table with each string value is probed first. If the probe fails (meaning this is the first time this categorical value is seen), a new *RIDSet* object is allocated. The string value is stored in the new *RIDSet* (*RIDSet*.value) and the *RIDSet* count (*RIDSet*.count) is set to one. *RIDSet*.count will eventually indicate the number of records that share this categorical value. The new *RIDSet* is then stored in the hash table, indexed by the string value. If the initial probe of the hash table does not fail, we will get back an existing *RIDSet* whose string value is the same as the current record. In this case, the *RIDSet*.count is simply incremented by one. In either situation, the current record's corresponding entry in the data array is set to point to this new *RIDSet*. Once the scan of the input data is finished, we will have exactly one *RIDSet* object per unique categorical value, and a fully initialized data array consisting of pointers to these *RIDSet*s. At this point, the original categorical data can be discarded. Figure 5 shows this state for a categorical *DataColumn* constructed from the ***Make*** attribute of our example dataset. Observe that if the data array is scanned in order and the string value of each referenced *RIDSet* is examined, the values seen will correspond exactly with the original input data.

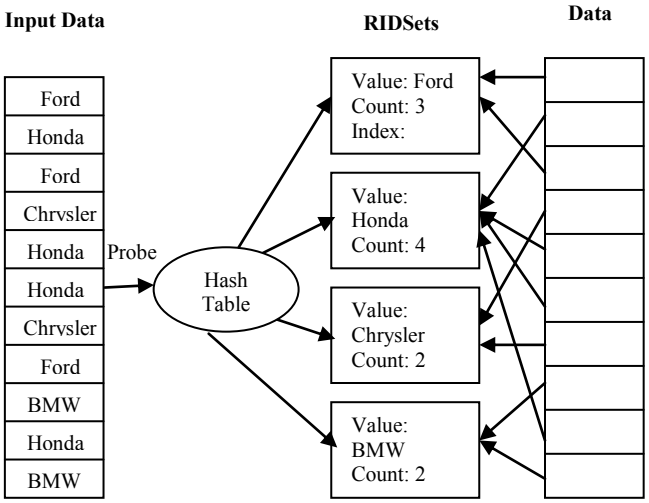

Figure 5. ***Make*** *DataColumn* (after input scan).

Now a sorted *RID-list* for this *DataColumn* must be built. As a performance optimization, rather than sort an entire *RID-list* as is done for numeric *DataColumn*s, each unique string value (or key) is collected and then sorted in a separate array. The number of keys should be smaller than the total number of records. This improves the sort cost. These keys can be retrieved from the *RIDSet*s, the hash table, or collected during the data-scanning phase. In our running example, an array of 4 keys would be sorted. Next, a temporary variable called *index* is allocated and its value is set to zero. We then step through the sorted array of keys and for each key, the corresponding *RIDSet* object is retrieved from the hash table and *RIDSet*.index is set to be equal to the *index*. The *index* is then incremented by *RIDSet*.count and *RIDSet*.count is set to be equal to zero. Once this scan is completed, the index value of each *RIDSet* will indicate where in the *RID-list* we store the RIDs of those records that share *RIDSet*.value. At this point, the sorted list of keys can be discarded.

The *RID-list* is now allocated and a scan of the *DataColumn*'s data array is initiated. For each *RIDSet* encountered, the corresponding record's RID is stored in the *RID-list* at the position: RID*Set*.index + RID*Set*.count. *RIDSet*.count is then incremented by one. Figure 6 shows the completed *DataColumn* after this scan has finished, along with the temporary sorted array of key values. Note that the *RID-list* is now in sorted order and all the *RIDSet*.counts have been restored to their original value.

3.1.2 DataGroup

DataGroup manages the *DataColumn* objects and is the mechanism through which the *ListRenderer* interacts with the data cache. It also manages two other crucial items - the *Restrictions* array and the *ResultSize* counter. The *Restrictions* array is an array of integers, indexed by RID that keeps track of the number of restrictions against each record. If a record's restriction count is zero, then that record is unrestricted and belongs to the current result set; otherwise it does not. The *ResultSize* counter indicates the number of unrestricted records (i.e., the size of the current result set).

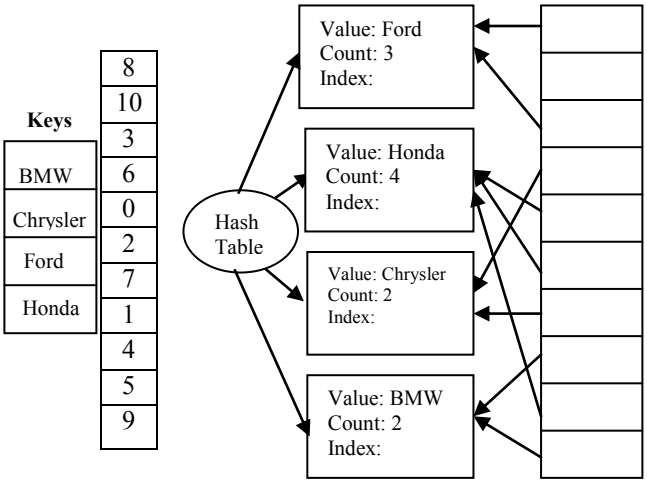

Figure 6. Categorical *DataColumn* for ***Make*** (completed).

Initially, all the restriction counts are set to zero and *ResultSize* is set to the total number of records. When a record is restricted along some attribute, its corresponding restriction count is incremented (even if that record is already restricted along some other attribute). Likewise, whenever a record is unrestricted along some attribute, its restriction count is decremented. Additionally, whenever a record moves out of or into the result set (i.e., its restriction count is incremented from zero or decremented to zero), the *ResultSize* counter is decremented or incremented accordingly.

3.2 Performance Restrictions

We now discuss how the system uses the data structures just described to realize instantaneous response time for doing interactive exploration. As mentioned earlier, every change to an

attribute control in the GUI represents a change in state. A state change is always a tightening or relaxation of the restriction bounds on one attribute. The *ListRenderer* passes down state changes to the *DataGroup* object, which in turn passes the state change down to the appropriate *DataColumn* object for building the cache.

Numerical Restrictions

While the attribute controls in the GUI represent the current query state for the user, internally the current state is represented differently. The current state for numeric *DataColumn*s is represented by two values, *LowerIndex* and *UpperIndex*. These values identify a sub range in the *DataColumn*'s *RID-list*. Since the *RID-list* is sorted by attribute value, a sub range in the *RID-list* corresponds to a sub range in attribute value. Initially, *LowerIndex* is set to 0 and *UpperIndex* is set to *n-1*. A state change on a numeric attribute always means that either the *LowerIndex* or the *UpperIndex* may have to change.

Suppose in our car example, all attributes are currently unrestricted, meaning that *LowerIndex* and *UpperIndex* for the ***Distance*** *DataColumn* are currently set to *0* and *n-1* respectively. The state change ***Distance*** < *100* now arrives indicating that the *UpperIndex* must be changed. Initiating a scan backwards through the *RID-list* from the current *UpperIndex* position does this. For each RID encountered, we lookup its corresponding attribute value in the data array and compare it to the new upper bound of *100*. If the value lies outside the new boundary, we update the restriction information in the *DataGroup* and continue the scan. We stop once we reach a value that lies within the new boundary. The current scan position in the *RID-list* becomes the new value for *UpperIndex*.

Categorical Restrictions

For categorical attributes, state changes always involve a single categorical value being either restricted or unrestricted. For ease of exposition, assume that each *RIDSet* in a *DataColumn* has an additional Boolean flag that indicates whether or not that value is currently restricted. Together, these Boolean flags represent the current state for this attribute. When a state change arrives (e.g., ***Make*** - *{Ford}*, i.e., *Ford* is excluded), we retrieve the corresponding *RIDSet* from the hash table. We then scan the portion of the *RID-list* marked by *RIDSet*.index and *RIDSet*.count and update the *DataGroup* statistics accordingly. We also set the Boolean flag in the *RIDSet* to indicate the new state. Note that the *RIDSet* structures allow us to examine only those RIDs that are relevant to the restriction.

The processes described above are essentially identical when relaxing restrictions. The primary difference is in how the *DataGroup* statistics are updated and, for numeric *DataColumn*s, in what direction the *RID-list* is scanned. It should be emphasized that when performing restrictions, we need only look at the data of the attribute involved. Additionally, we only examine data for those records actually affected by the restriction change. If a restriction change only affects 50 records, then we only examine those 50 records in one *DataColumn*, regardless of the total number of records or attributes.

4. Conclusion

We presented a model of a database exploration engine that implements iterative querying in web-based database applications. It has fast response time even when exploring hundreds of thousands of records containing hundreds of attributes. Query borders in our proposal can also be made non-strict such that records that lie just outside the query region can still be included in the result set.

The speed at which query changes are processed is enhanced by processing changes immediately rather than waiting for the user to hit a submit button. This not only results in small query changes that can be effected quickly, but also offers the additional advantage of immediate feedback for the user.

References

[1] Dar, S., Franklin, M., Jonsson, B., Srivastava, D., and Tan, M. *Semantic Data Caching and Replacement*. Proceedings of VLDB 96. Mumbai, India, 1996.

[2] Date, C. *An Introduction to Database Systems*. Addison-Wesley. Seventh Edition. California. 2000.

[3] Elmasri, R. and Navathe, S. *Fundamentals of Database Systems*. Addison-Wesley. Third Edition. California. 2000.

[4] International Business Machines. *IBM Intelligent Miner* User's Guide, Version 1 Release 1, SH12-6213-00 edition, June 1996.

[5] Kossmann, D., and Stocker, K. *Iterative Dynamic Programming: A New Class of Query Optimization Algorithms*. ACM Transactions on Database Systems. Volume 25, Issue 1. March 2000.

[6] Zloof, M. *Query-By-Example: The Invocation and Definition of Tables and Forms*: In Douglas S. Kerr, editor, Proceedings on the International Conference on Very Large Data Bases, pp. 1-24. ACM. 1975.

[7] http://www.excite.com. 2001.

[8] http://www.yahoo.com. 2001.

Biography

Ramzi A. Haraty is an Assistant Professor of Computer Science at the Lebanese American University in Beirut, Lebanon. He received his B.S. and M.S. degrees in Computer Science from Minnesota State University - Mankato, Minnesota, and his Ph.D. in Computer Science from North Dakota State University - Fargo, North Dakota. His research interests include database

management systems, artificial intelligence, and multilevel secure systems engineering. He has well over 40 journal and conference paper publications. He is a member of Association of Computing Machinery, Arab Computer Society and International Society for Computers and Their Applications.

Mazen Hamdoun is currently pursuing his Masters of Science in Computer Science at the Lebanese American University.