Continuous querying in database-centric Web applications

John C. Shafer *,1, Rakesh Agrawal 1
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA

Abstract

Web applications are becoming increasingly database-centric. Unfortunately, the support provided by most Web sites to explore such databases is rather primitive and is based on the traditional database metaphor of submitting an SQL query and packaging the response as an HTML page. Very often, the result set is empty or contains too many records. It is up to the user to refine the query by guessing how the query constraints must be tightened or relaxed and then go through another submit/response cycle. Furthermore, once results are displayed, typically no further exploration capabilities are offered. Web applications requiring interactive exploration of databases (e.g. e-commerce) need that the above submit/response metaphor be replaced with a continuous querying metaphor that seamlessly integrates querying with result browsing. In addition to supporting queries based on predicates on attribute values, queries based on example records should also be supported. We present techniques for supporting this metaphor and discuss their implementation in a Web-based database exploration engine.

Keywords: Search; Database; Interactive

1. Introduction

Web applications are becoming increasingly database-centric. In a 1997 Forrester survey [5], respondent companies indicated that nearly 40% of the content at their Web sites originated from databases. This was expected to rise as high as 65% by 1998, and that this fraction was expected to increase. Many new Web applications require that a user be able to interactively explore these databases over the Internet or an internal network.

A common example of such interactive exploration is the task of finding products or services matching a user’s requirements. While this is a widely performed task [9], the support provided by current Web sites for implementing this functionality is rather primitive. Typically, a server-side database is relied upon for all query processing. The user is presented a form for providing specifications of the desired product in terms of bounds on the values of the product attributes (e.g. a 3.3 V zero delay clock buffer in 16-pin 150-mil SOIC or TSSOP package with output skew less than 250 ps and device skew less than 750 ps having an operating range of 25–100 MHz). On submission, this information is used to construct an SQL query that is in turn submitted to a server-side database. The result is returned to the browser formatted as an HTML page. Very often, the result set is empty or contains too many records. It is up to the user to refine the query by guessing how the query constraints must be tightened or relaxed and then go through another submit/response cycle. Furthermore, once results are displayed, typical...
cally no further exploration capabilities are offered. As a result, the user needs knowledge of not only the domain of interest but also the particular dataset. Further aggravating this problem is that the round-trip time between browser and the database server for each submit/response cycle is often frustratingly large.

The problem is that database query technology is targeted at reporting rather than user exploration. In traditional database applications, queries are rigid in that they are intended for asking very specific questions. The query results are interesting regardless of whether they contain zero records or ten thousand. In a sense, an individual query itself is the goal. In user exploration, the goal is not simply an individual query or its results, but rather locating particular records of interest. Rarely can this be achieved with a single query. As a result, users typically issue many related queries before they are finally satisfied.

What is needed is that this ‘submit a query and wait for a response’ metaphor be replaced by a new continuous querying metaphor. The user should be able to combine searching with result browsing so that the user simultaneously sees the current query and the qualifying records in a single view. As the user changes the query constraints, the user should immediately see the impact on the qualifying records in that view.

We present techniques for supporting this continuous querying metaphor. These techniques have been implemented in a database exploration engine, we call Eureka. We present in Section 2 the user interface that facilitates database exploration using the continuous querying metaphor. For this metaphor to be successful, it is imperative that as soon as a user manipulates a GUI control, the user sees its effect instantaneously, which in turn requires well-tuned data structures. In Section 3, we present the design and implementation of the Eureka engine. We conclude with a summary and some possible directions for future work in Section 4.

1.1. Related work

A large number of e-commerce sites provide parametric search capabilities in which users search for desired products by providing bounds on attribute values. Stock screens at investment sites such as Charles Schwab, travel package selection at travel sites such as Travelocity, electronic component search at semiconductor sites such as Cypress are examples of this type of search. Some e-commerce tools (e.g. Net.Commerce [7]) provide support for implementing such searches. As stated earlier, these sites typically rely entirely on a server-side database for query processing. They are thus limited to the submit/response metaphor and suffer from the problems of long response times and too many or too few answers.

Some newer sites are providing a subset of interactive exploration capability described in this paper. For example, Microsoft’s Carpoint, Cars.com and Wireless Dimension combine browsing with querying; as the query is changed, the user immediately sees the effect on results. These sites currently do not support querying based on example products. The details of their implementations are not available in published literature. It is doubtful that the Wireless Dimension’s Javascript implementation (which uses HTML for its output) is designed to scale to large product sets, and only Carpoint (using native ActiveX code) allows users to explore datasets with more than a few hundred products.

An interesting approach to handle the problem of too many or too few answers was taken by 64K Inc. [1]. Although still an HTML form-based approach, the query pages generated by the 64K engine contain histogram information, showing how records are distributed over each attribute’s range of values, as well as a count of the total number of records. This information is meant to provide hints to the user for modifying the query before resubmitting it. If a query results in too many records, rather than showing them to the user, the engine redisplay the query page updated with new histogram and count information. In the case of too few answers, the engine uses domain-specific distance metrics to relax the query and return nearby records. However, the 64K search metaphor is still the standard submit/response metaphor (although perhaps with fewer cycles and richer features).

2 www.schwab.com
3 www.travelocity.com
4 www.cypress.com
5 carpoint.msn.com
6 www.cars.com
7 www.wirelessdimension.com
Another approach to handling too many answers is represented by the FOCUS application described in [10]. In this technique, the results of a query are cached and displayed in a compressed table. Further restrictions to reduce the result set are applied on the cached results. However, as the authors acknowledge, FOCUS is mainly suited for tables with up to a few hundred records and attributes. In the case of Spotfire Pro, results are presented in various graphical formats (e.g. pie charts, scatter plots, etc.) and users can manipulate sliders and list boxes to interactively search through the data. Spotfire Pro is a client-side application that must be installed locally on the user’s machine. Details of their implementation are not available in the published literature.

Other related work includes the dynamic query and starfield work of [2] and [3]. Like Spotfire Pro, users can manipulate numeric sliders and other GUI controls and see the effects of these actions on result sets displayed in 2D scatter plots. This work focuses mainly on the human-interface aspect of this search metaphor and the implementation details are rather sparse.

2. User interface

Fig. 1 shows Eureka’s user interface. Records are displayed in a scrollable list format with a separate column for each numeric and categorical attribute. 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.

Note that 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 in the upper left 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 continuous querying model with the conventional form-based approach. The user would be entering her requirements into a static query page and would not know the result of her actions until
she submitted her query and received the response. If her query is too restrictive, she would most likely be presented with a short message telling her that no records matched her criteria, and that she should try again. If the query is too loose, she 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 she should relax or tighten her query, forcing her to repeat this submit/response cycle as she adjusts her requirements unaided. Each such cycle requires the attention of the Web and database servers, not to mention a round-trip through the network. In the case of continuous 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 in Eureka, this allows a query-by-example 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. In our car example, a user who does not know about horsepower or what a ‘fast’ 0–60 mph 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.

As shown in Fig. 2, the user indicates the example records by highlighting the corresponding rows with the mouse (in this case, a Porche 968 and a Pontiac Firebird). The attribute space is then collapsed around those examples by holding down the control key and using the mouse to click on the headings of the columns to collapse. This action immediately results in the attribute controls being adjusted to reflect the new query space. Simultaneously, the result pane is updated to reflect the new qualifying records. The user can now continue exploration by explicitly manipulating attribute controls or she can proceed by selecting some other example records and collapsing the attribute space further.

2.2. Exploration based on similarity

In addition to querying by example, Eureka allows ‘sorting by example’ as well. This is essentially similarity search, except that instead of treating similarity as a separate stand-alone query like Excite does for Web pages, it is integrated into Eureka’s existing view. After selecting one or more sample records, the user can click the rank button (see Fig. 3) and sort all the records by how ‘near’ they
are to the selected examples. Nearness is based on a similarity model that computes the weighted sum of the normalized difference in attribute value [6,11]. The similarity scores are then displayed in a new column labeled ‘Rank’ which appears after the rank button is pressed.

The user can also control similarity by manipulating the select buttons that appear underneath each column. Normalized difference is represented by the ‘near’ option. Selecting ‘don’t care’ removes an attribute from the similarity model entirely. The ‘same’ option rewards identical values with a 100% score for that attribute — all other values get 0%. For numeric attributes, we have included additional support for ‘directional nearness’. As shown in Fig. 3, we can specify that larger values (the ‘more’ option) or smaller values (the ‘less’ option) should not be penalized in the scoring model. In our example, this allows us to increasingly penalize cars with 0–60 mph times larger than 9 s, but not penalize those with smaller times.

3. System design and implementation

We now describe the details of how we implement continuous querying in Eureka. In order to obtain interactive response times, we exploit several key observations. First and foremost is that 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 non-trivial 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 in Eureka 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 described in [4] can be used.

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 at hand 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.

Eureka also takes 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, Eureka need only 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 instant. Since query adjustments are always effected immediately on the result set, these changes are always small and along a single attribute.

The last important observation is that, regardless of the result size or the capability of scrolling through those results, the user can only see those records currently displayed on the screen, the full result set is never completely visible. As a result, Eureka never explicitly generates a complete list of all records satisfying the current query state. In fact, the list of records Eureka does create never contains any records other than those currently being displayed. By taking advantage of these observations, Eureka is able to exhibit extremely fast response times, even with datasets containing hundreds of attributes and hundreds of thousands of records.

3.1. Architecture

Fig. 4 gives a high-level overview of the Eureka architecture. The DataColumn objects represent the data cache with each DataColumn representing one column of attribute data. DataGroup maintains this cache and is responsible for managing statistics needed to generate the result sets. 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 be forced 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 Eureka 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 API calls. While the ListRenderer is not responsible for much of the engine logic, it is nevertheless an important component, as the back-end is specifically designed with assumptions about how the user will interact with the system.

The final piece labeled Eureka encompasses the entire design. In our 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 or perhaps simply by the user clicking on an URL in a Web browser (e.g. a link labeled ‘Explore mid-cap stocks in Eureka’). The Eureka component is also responsible for retrieving the data from the data source. It does so through a standard API called the ‘DataPump API’. 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 Fig. 4, the DataPump communicates via HTTP to a Java servlet running on a Web server. This servlet uses JDBC to communicate with the database. When Eureka requests data via the DataPump API, the DataPump object passes the request to the servlet,

![Fig. 4. Architecture overview.](image-url)
which in turn issues a query to the database. Data is returned to Eureka from the DataPump in column-order rather than row-order using callbacks. This allows each column to be converted into DataColumn objects independently and asynchronously by using multiple threads to service the callbacks. 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.

In other situations, such as intranet applications, we have incorporated the JDBC communication directly into the DataPump object. We have also built a file-based DataPump that can load data from a local comma-delimited file. Regardless of the particular DataPump implementation, it should be noted that Eureka, DataPump, ListRenderer, DataGroup and the DataColumn objects all reside on the client machine and not on a server. For the remainder of this discussion, we will focus on the core engine pieces consisting of the ListRenderer and the components underneath it.

For concreteness, we will be describing the Eureka design using the example dataset shown in Fig. 5. 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. Note that this example demonstrates how the data received and cached by Eureka 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.

### 3.1.1. DataColumns

Each DataColumn is responsible for caching and indexing one attribute’s worth of data. DataColumns consist of two basic components: an array of data values indexed by record id, and a rid-list. The rid-list contains rids sorted by their corresponding attribute value. As we shall see in Section 3.2, the rid-lists 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 and so are described separately below.

**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 Fig. 6.

<table>
<thead>
<tr>
<th>rid</th>
<th>Ford</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Honda</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Ford</td>
<td>537</td>
</tr>
<tr>
<td>3</td>
<td>Chrysler</td>
<td>117</td>
</tr>
<tr>
<td>4</td>
<td>Honda</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>Honda</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Chrysler</td>
<td>363</td>
</tr>
<tr>
<td>7</td>
<td>Ford</td>
<td>192</td>
</tr>
<tr>
<td>8</td>
<td>BMW</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>Honda</td>
<td>89</td>
</tr>
<tr>
<td>10</td>
<td>BMW</td>
<td>207</td>
</tr>
</tbody>
</table>

**Fig. 5. Example dataset.**

<table>
<thead>
<tr>
<th>RID List</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>537</td>
</tr>
<tr>
<td>8</td>
<td>117</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>363</td>
</tr>
<tr>
<td>7</td>
<td>192</td>
</tr>
<tr>
<td>10</td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td>89</td>
</tr>
<tr>
<td>2</td>
<td>207</td>
</tr>
</tbody>
</table>

**Fig. 6. Distance DataColumn.**
Observe that $Data[RidList[i]]$ gives the value of the $i$th item in the sorted list of Data values.

**Categorical DataColumns.** Categorical DataColumns are somewhat more involved than numeric ones. In addition to a data array and a rid-list, categorical DataColumns also require a hashtable and specialized objects called RidSets. 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. The importance of these additional data structures in supporting fast restrictions will be explained in Section 3.2.

We first describe the creation of the categorical data array. Instead of allocating an array strings to store the data values, we allocate an array of RidSet references. We also allocate an empty hashtable. As we scan the incoming categorical attribute data, we first probe the hashtable with each string value. If the probe fails (meaning that this is the first time we have seen this categorical value), we allocate a new RidSet object. 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 hashtable, indexed by the string value. If our initial probe of the hashtable does not fail, we will get back an existing RidSet whose string value is the same as the current record. In this case, we simply increment RidSet.count by one. In either situation, we then set the current record’s corresponding entry in the data array 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 RidSets. At this point, the original categorical data can be discarded. Fig. 7 shows this state for a categorical DataColumn constructed from the *Make* attribute of our example dataset. Observe that if we now scan the data array in order and examine the string value of each referenced RidSet, the values seen will correspond exactly with the original input data.

Now we must build a sorted rid-list for this DataColumn. As a performance optimization, rather than sort an entire rid-list as is done for numeric DataColumns, we instead collect each unique string value (or key) and sort them in a separate array. The number of keys should be smaller than the total number of records and so will improve our sort cost. These keys can be retrieved from the RidSets, the hashtable, or collected during the data-scanning phase. In our running example, we would sort an array of four keys. Next, we allocate a temporary variable called *index* and set its value to zero. We then step through the sorted array of keys and for each key, we retrieve the corresponding RidSet object from the hashtable and
set RidSet.index equal to index. We then increment index by RidSet.count and then set RidSet.count equal to zero. Once this scan is completed, the index value of each RidSet will tell us where in the rid-list the rids of those records that share RidSet.value to store. At this point, the sorted list of keys can be discarded.

The rid-list is now allocated and we initiate a scan of the DataColumn’s data array. For each RidSet encountered, we store the corresponding record’s rid in the rid-list at the position: RidSet.index + RidSet.count. We then increment the RidSet.count by one. Fig. 8 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

As we will see later, the 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, i.e. the restrictions array and the resultSize counter. The restrictions array is an array of integers, indexed by record id, 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).

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. Further details about this process, including illustrating examples, are given in Section 3.2 below.

### 3.2. Performing restrictions

We now discuss how Eureka 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 Eureka GUI represents a change in state. A state change is always a tightening or relaxation of the restriction bounds on one attribute. State changes are
passed down by the ListRenderer to the DataGroup object, which in turn passes the state change down to the appropriate DataColumn object for building the cache.

### 3.2.1. Numeric 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 DataColumns is represented by two values, `lowerIndex` and `upperIndex`. These values identify a subrange in the DataColumn’s rid-list. Since the rid-list is sorted by attribute value, a subrange in the rid-list corresponds to a subrange 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, that 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. This is done by initiating a scan backwards through the rid-list from the current `upperIndex` position. For each rid encountered, we look up 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`. This process is illustrated in Fig. 9.

### 3.2.2. 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 hashtable. 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. Fig. 10 shows how the data structures would look after performing both the restrictions `Distance < 100` and `Make = [Ford]`. 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 DataColumns, 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.

### 3.3. Rendering the list

After a restriction change has been processed, the ListRenderer may be notified to update the result set information displayed in the GUI. This event is triggered whenever at least one record moves in or out of the current result set. However, as mentioned earlier, the user can only see the portion of the result set currently displayed on the screen. Thus the ListRenderer never need draw (and thereby instantiate) the entire result set. All it has to do is

redraw the 20 or so records that the user will see at any given moment.

Note that what appears in the Eureka GUI to be a standard scrollable list displaying matching records is in fact a separate paintable ‘canvas’ and an independent scrollbar. If this component was implemented as a standard GUI list object, then we would be forced to insert every matching record from the current result set into that list. Changing sort order would also require reinserting the entire result set. While this may be satisfactory for small datasets of a few hundred records, this does not scale to datasets with thousands or tens-of-thousands of items.

To display the current visible portion of the result set, the ListRenderer needs a list of those records it is to display. Once it has a list of rids, the ListRenderer can interrogate the DataGroup object for the attribute data needed to render those records on the screen. This list of rids must reflect the current result set, the current sort order, and the current position within the result set (as indicated by the scrollbar).

Rendering begins with the ListRenderer making an API call to the DataGroup asking for N unrestricted rids, starting at position P and sorted by column C. Upon receiving this request, the DataGroup object passes it on to the corresponding DataColumn. The DataColumn initiates a scan of its rid-list, starting at the indicated position and in the appropriate direction (depending on an increasing or decreasing sort order). For each rid scanned, its restriction count is examined; if the count is zero, the rid is added to the list of rids to be returned. The scan halts when enough rids have been collected, or there are no more rids to scan. For categorical DataColumns, we perform an optimization whereby if a rid is restricted, we examine its corresponding RidSet to see if that RidSet is restricted. If so, we use its count and index information to skip that entire block of rids in the rid-list. Due to realistic limits on the size of the GUI display, the number of requested rids will likely never be larger than about 30, making this scanning process extremely fast. Note that if the GUI requests rids in record id order, then the DataGroup can handle the request itself by scanning the restrictions array directly.

In our car example, assume that we have the two restrictions illustrated in Fig. 10 and our current displayed position is at the very top of the result set. Assuming our sort column is Distance and the GUI is sized such that only three records are visible at any given instant, then the rid-list returned to the ListRenderer would be 1, 5 and 4. If our sort column is Make, then the rids would be 8, 1 and 4.

![Fig. 10. Example categorical restriction (Make = Ford).](image)

3.4. Memory requirements

The following formula gives the estimate of the memory requirements for the cache. In the formula, \( n \) is total number of records, \( f_n \) is the number of numeric attributes, \( f_c \) is the number of categori-
cal attributes and $v_c$ is the total number of unique categorical values.

\[
((f_n + f_c) \times n \times (\text{data} + \text{rid})) + (f_c \times \text{hashtable})
+ (v_c \times \text{ridsets}) + (n \times \text{restriction}).
\]

Each data-array element in a DataColumn consumes 'data' bytes and each rid-list element consumes 'rid' bytes. From the categorical DataColumn, 'hashtable' represents the average number of bytes required for hashtables and 'ridsets' represents the average size of a RidSet (including its associated string value). Finally, 'restriction' is the number of bytes consumed by each entry in the DataGroup’s restrictions array.

In our Java implementation, 'data' is 4 bytes. Depending on the value of $n$, 'rid' may either be 2 bytes (for up to 64 thousand records) or 4 bytes (up to 4 billion records). In virtually every situation, the restrictions array can be implemented with single byte values (for up 255 attributes), meaning that 'restriction' is 1 byte. For example, the dataset used for illustration in Section 2 contained 3342 records, with each record consisting of 9 attributes. The cache requirement for this dataset is less than 200 KB.

4. Conclusions

We presented the design of Eureka, a database exploration engine that implements continuous querying in data-centric Web applications. Eureka has extremely fast response time even when exploring hundreds of thousands of records containing hundreds of attributes. Besides predicates on attribute values, Eureka also supports searching using example records. Query borders in Eureka can also be made non-strict such that records that lie just outside the query region can still be included in the result set.

Obtaining interactive response times in Eureka is made possible by exploiting several key observations. One of the primary observations exploited is that searching is a process and not a one-step operation. As a result, users invariably issue multiple ‘queries’ over the data which are typically not unrelated. Each query in fact is usually some modification of a previous one. When standard database technology such as SQL is used, this fact is not exploited; although query state may be maintained in the HTML search form, each query submitted to the database is treated as a new query regardless of how small the change. By exploiting the concept of state and integrating it into the engine’s data structures, Eureka is able to handle these successive queries extremely quickly.

The speed at which query changes are processed is further 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.

The final optimization we leveraged is that Eureka is a visual tool rather than a reporting tool. As such, full result sets never need full instantiation. Since users can only view as many result records as can fit on the screen at one time, the subset of the results that Eureka does instantiate is never larger than 20 or so records.

The design we presented here for Eureka is our second iteration on building a database exploration engine. In our first implementation, we followed the conventional submit/response metaphor. However, instead of directly returning the result of the SQL query, we first examined the size of the result set. If the result set was empty, we used a query relaxation algorithm that removed one constraint at a time until we obtained a non-empty set to return to the user. On the other hand, if there were too many result records, we selected a ‘representative’ subset of the result set and returned only those records instead. The representative subset can be determined by clustering the result records and selecting the medoids of the clusters [8]. In practice, choosing the representatives by a random sampling of the result set worked equally well and was much faster. The user could then explore the neighborhood of a record so returned by optionally specifying in the GUI the search direction for each attribute (e.g. faster processor, more memory, etc.) These user hints were then used to generate modified SQL and the cycle was repeated.

While technically interesting, we found that the system became quite painful to use because of the delay the user encountered between two successive
interactions. Some users also found it unnerving that we were only showing representative records; there was a nagging fear that they might have missed something important.

The current Eureka design addresses these concerns effectively. The user explores the neighborhood by moving attribute controls and there is instantaneous response that the user can see. Indeed, we have received very positive feedback from the users of Eureka in an internally deployed Web application.

Acknowledgements

We would like to thank Sunita Sarawagi who helped design and implement our first database exploration engine. We also thank Andreas Arning, Daniel Gruhl, Dimitrios Gunopulos, Howard Ho and Magnus Stensmo for their contributions to the discussions.

References