Research Report
Design and Evaluation of Incremental Data Structures and Algorithms for Dynamic Query Interfaces

Egemen Tanin[*] Richard Beigel[†] Ben Shneiderman[‡]

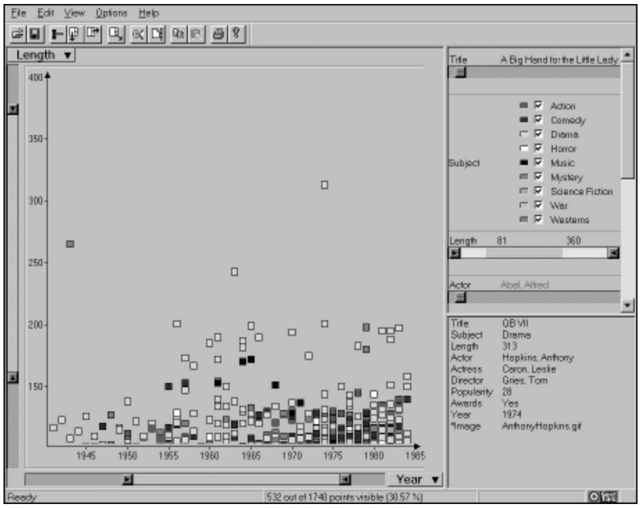
Human-Computer Interaction Laboratory[§]
Department of Computer Science
University of Maryland

Abstract *Dynamic query interfaces* (DQIs) are a recently developed database access mechanism that provides continuous real-time feedback to the user during query formulation. Previous work shows that DQIs are an elegant and powerful interface to small databases. Unfortunately, when applied to large databases, previous DQI algorithms slow to a crawl. We present a new incremental approach to DQI algorithms and display updates that works well with large databases, both in theory and in practice.

Keywords Data Structure, Algorithm, Database, User Interface, Information Visualization, Direct Manipulation, and Dynamic Query.

1. Dynamic Querying

Dynamic query interfaces (DQIs) are a recently developed mechanism for specifying queries and visualizing their results [1, 2, 5, 6, 7, 9, 8, 12, 14]. Unlike textual query languages such as SQL, DQIs are graphical. A great advantage of DQIs is that they provide continuous feedback to the user as the query is being formulated. Experiments have shown that querying with DQIs is faster, easier, more pleasant, and less error-prone than with other querying interfaces [2, 14]. A sample DQI is presented in Figure 1 (created by [8]).



Title A Big Hand for the Little Lady

Subject
☑ Action
☑ Comedy
☑ Drama
☑ Horror
☑ Music
☑ Mystery
☑ Science Fiction
☑ War
☑ Westerns

Length 81 360

Actor Abel, Alfred

Title QB VII
Subject Drama
Length 313
Actor Hopkins, Anthony
Actress Caron, Leslie
Director Gries, Tom
Popularity 28
Awards Yes
Year 1974
*Image AnthonyHopkins.gif

Ready

532 out of 1740 points visible (30.57 %)

Figure 1. Spotfire: a good example for DQIs (www.ivee.com). The user specifies a query using the widgets along the left, bottom, and right of the display. The hit set for the current query is displayed as a starfield on the left. This example does not contain a bar giving aggregate information about the database but the hit set size is shown as a count at the bottom.

Queries are made using widgets, such as range sliders (for continuous data attributes), alphanumeric sliders (for textual attributes), toggles (for binary attributes), and check boxes (for discrete multi-valued attributes), to specify each attribute (dimension) of the data. Output is provided via a starfield display (a 2-dimensional projection of the set of hits), bars (such as a preview bar that displays the number of

[*] egemen@cs.umd.edu, partially supported by NASA grant NAG 52895.

[†] beigel@cs.umd.edu, partially supported by NSF grants CCR-8958528 and CCR-9415410, and by NASA grant NAG 52895, on sabbatical from Yale University until 8/1/97.

[‡] ben@cs.umd.edu, partially supported by NSF grants EEC-9402384 and IRI-9615534, and by NASA grant NAG 52895, also affiliated with the Institute for Systems Research.

[§] address: Department of Computer Science, U. of Maryland at College Park, College Park, MD 20742, USA, for more information: http://www.cs.umd.edu/projects/hcil.

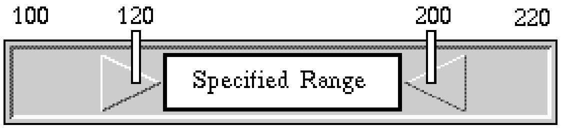


Figure 2. A sample range slider. By moving the arrows, the user specifies a range, which is represented by the white rectangle. The numbers above the arrows give the current range. The numbers on the far ends of the range slider are the extreme values that the attribute can take.

hits), and charts (which provide other aggregate information). The widgets are tightly coupled: as the hit set varies all the widgets are updated to show the hit set's bounding rectangle, so the widgets provide a limited form of output as well. If desired, we can even display a histogram on each widget to show the distribution of data in its dimension. The user may click on an individual point on the starfield for "details on demand".

Range sliders are used to manipulate continuous attributes. See Figure 2 for a sample range slider. A range slider contains a pair of arrows, one at each end. The user selects a range slider by clicking on it, and the user adjusts the range by dragging either arrow with the mouse. As the range is being adjusted, the starfield, bars, and charts are updated. Histograms and states of the other widgets can also be updated. Thus, for each tiny increment of the range slider, much information must be computed rapidly.

Toggles allow the user to specify a binary attribute of the data. On the display they look like boxes. Internally they can be implemented directly without much trouble or treated as a nearly trivial special case of range sliders. List boxes, radio buttons, and various other discrete widgets can be handled with similar ease.

Alphanumeric sliders allow the user to specify a range of strings. Although our auxiliary data structures apply to them as well, the fine granularity of alphanumeric data seems to necessitate additional implementation ideas that are best described in a separate paper.

We propose a new approach to DQI algorithms that can handle larger databases than previous implementations. This paper expands our previous note [13] by providing a detailed explanation and evaluation of our DQI algorithms. We present our approach in general in Section 2. We give a detailed explanation of the data structures and algorithms in Section 3. We analyze the complexity of our algorithm in Section 4. We evaluate our approach experimentally in Section 5. We state our conclusions and future work in Sections 6 and 7.

2. The Incremental Approach

In DQIs queries are formed in an incremental fashion. For example: to set a range on a slider, the user drags the two arrows of the slider to desired positions on the display. This enables the user to visualize intermediate results until a desired final set of positions is reached. Also, a query can be formed via a conjunction (or disjunction) of constraints on more than one or two attributes. This also produces numerous intermediate results to be displayed. Therefore, we chose the incremental query formulation paradigm in our designs for algorithms and data structures for DQIs.

The incremental approach gains its efficiency from the following innovations:

Active subset We define an "active" subset of the database, of limited size, which we store in main memory. (While in principle the size of main memory may seem like a severe limitation, in practice DQI algorithms seem to be limited more by time than by space).

Auxiliary data structures We augment the active subset with data structures that facilitate continuous querying (users can tolerate a response time of about 0.1 seconds for continuous operations [1]).

Reprocessing The auxiliary data structures change only when the user clicks on a widget. After such an action the user will accept a delay of approximately 1 second or less, during which we reconstruct the auxiliary data structures.

Incremental display Slight changes in the query tend to cause only slight changes in the output. By computing and displaying the difference, we can update the display continuously.

We envision using the DQI algorithms in tandem with a *query previewer* [3] that allows the user to browse a huge database and select a manageably small subset to scan. Once the user selects such a subset, the query previewer passes its bounding rectangle to the DQI, which then takes control. The bounding rectangle for the active subset determines the extreme values for each attribute. Therefore, the query preview approach can also be considered as an application of the incremental querying paradigm (in a more broad sense). If, at some later time, the user wants to look outside the active subset, then the simplest solution is for the DQI to return control to the query previewer. This will be considered in a future paper on the interaction between the DQI and the query previewer. The data structures and algorithms are given in the following section with an example on range sliders.

3. Data Structures and Algorithms

Our DQI algorithms perform three major operations: *setup*, *selection*, and *querying*.

Setup occurs when the query previewer passes control to the DQI. During setup, the widgets, starfield display, bars, and charts are initially drawn on the display. The DQI reads the active subset. In addition, it makes a copy of the active subset and re-scales each attribute to the range $[1, p]$ where p is the number of pixels in the attribute's range slider. This step is important because we need this re-scaled information frequently. Because setup occurs infrequently, we can allow several seconds for it.

Selection occurs when the user clicks on a range slider. During selection, the algorithm computes the auxiliary data structures, which depend on the currently selected attribute and the current ranges for the other attributes. Various experiments with user interfaces show that we must respond to the mouse click in about 1 second in order not to annoy the user. At the cost of a factor of 2 in memory, we could precompute the auxiliary data structures whenever the mouse is moved close to a new range slider. Using somewhat more memory, we could steal cycles from the query operation in order to precompute the auxiliary data structures for several sliders. Thus 1 second may be an overly conservative bound on the time for selection.

Querying occurs continuously as the user drags the mouse to update the selected range slider. During querying, the algorithm recomputes the hit set and updates the output on the display. For the purpose of timing, we say that a single query occurs each time the DQI detects a change in the position of the mouse on the range slider. Experiments show that DQIs must process each query in about 0.1 seconds in order to give a continuous response [1].

During the selection operation, the DQI computes the *maximum hit set*, which is the hit set determined by the extreme values for the selected attribute and the current ranges for the other attributes. Then it partitions the maximum hit set into p buckets, one for each user-specifiable value for the current attribute. This is essentially a linear time counting sort of the maximum hit set. We store the size of each bucket and all left-to-right partial sums of these sizes (an example range slider with bucket information included is given in Figure 3).

In selection, we also compute the information that facilitates computation of histograms and tight coupling of range sliders, i.e., the redisplay of all slider ranges when any range is changed. To achieve this goal a two dimensional array for each slider is kept (i.e., size p^2, shown in Figure 4).

Thus, any time the range slider is updated, even by a large number of pixels (as might happen if the user moves the mouse extremely fast), we can determine the number of hits in constant time. We can also determine ranges for the other sliders in constant time per slider, and histograms for the other sliders in constant time per point. Changes to the display are determined by scanning the buckets between the older attribute value and the new one. The display is updated in time that is linear in the number of changes.

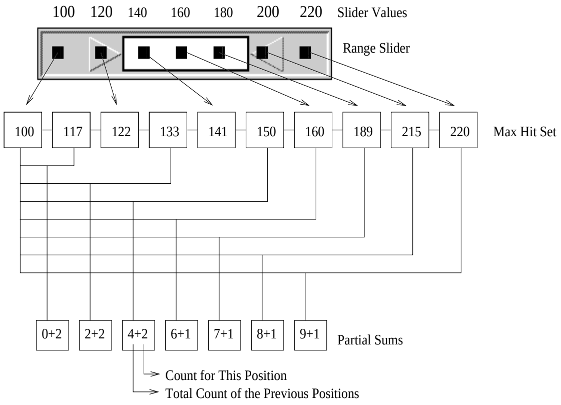
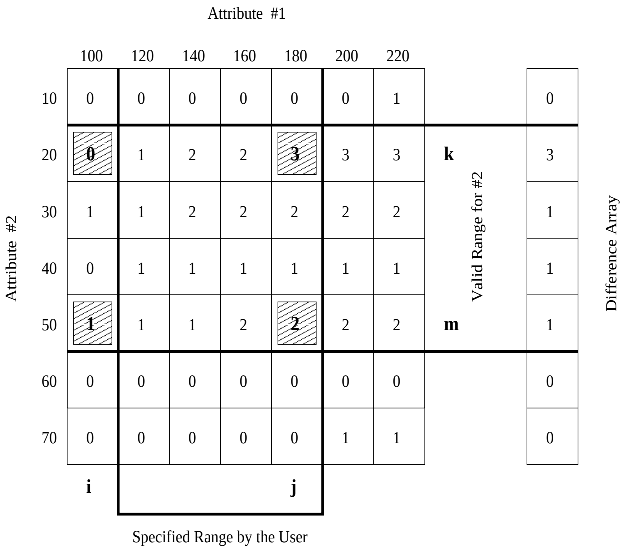


Figure 3. A sample range slider with bucket information. This example shows 10 records distributed on the range slider sorted in ascending order for the related attribute. The arrows from the range slider to the hit set show the related positions in the sorted maximum hit set that each discrete pixel position of the range slider maps to where the numbers in this hit set give the values of the records for that attribute represented by the slider. The lower array of boxes represents the partial sums of counts for a given pixel position of the range slider. This example contains only 7 hypothetical pixels for simplicity.

If the user changes the axis parameters for the starfield display, then the hits are redisplayed (projected on the new pair of coordinates) but none of the internal data structures is changed. The only thing that is updated is the starfield information.

The starfield information is equivalent to a two dimensional "dirty-pixel" array where for each starfield pixel we keep a count. Whenever a pixel count reaches zero that pixel color is set to the background color (and visa versa i.e., if it was zero and becomes a positive integer then we plot that pixel on the starfield display using the foreground color). Programmers can use these counts to give another form of histogram information to the user. Also, they can immediately find the necessary pixels on the starfield display that must be up-



	100	120	140	160	180	200	220		
10	0	0	0	0	0	0	1		0
20	▨ 0	1	2	2	▨ 3	3	3	k	3
30	1	1	2	2	2	2	2		1
40	0	1	1	1	1	1	1		1
50	▨ 1	1	1	2	▨ 2	2	2	m	1
60	0	0	0	0	0	0	0		0
70	0	0	0	0	0	1	1		0
	i				j				

Figure 4. A table containing the histogram and the tight coupling information. This figure uses the same data and example as Figure 3. Each box in the square holds a count. The user sets the range for attribute 1. For a given specified range of attribute 1 we can find the valid range for attribute 2 by projecting the hit set's bounding rectangle onto attribute 2. Each row is a prefix sum of counts from left to right. So if we subtract column j from column i we can use the resulting difference array to find the valid range for attribute 2. The highest nonzero row (k) and the lowest nonzero row (m) give the valid range for attribute 2. Note that the histogram information for attribute 2 is just the difference array.

dated (i.e., only the pixel positions where the counts vary).

4. Theoretical Complexity

Let r denote the number of records in the active subset, a the number of attributes, and b the number of bytes needed to store the value of a single attribute. Let p denote the length in pixels of each range slider, f the area in pixels of the starfield, and u the average number of pixels that need to be updated in the starfield display per query (this number depends in a nontrivial way on the size of the starfield, the velocity of the range slider, and the clustering of data in the active subset). Let m denote the number of records in the maximum hit set.

The active subset occupies $r \cdot a \cdot b$ bytes. The rescaled active subset occupies $O(r \cdot a)$ bytes. The bucket partition also occupies $O(r \cdot a)$ bytes. The data structures for tight coupling occupy $O(a \cdot p)$ bytes. The data structures for range histograms occupy $O(a \cdot p^2)$ bytes. The starfield occupies f bytes.

Setup takes time $O(r \cdot a \cdot b)$.

There are four components to the time for selection. Determining the maximum hit set takes time $O(r \cdot a)$. Sorting the maximum hit set takes time $O(m)$ (there is no log factor because we discretize the data). Computing the auxiliary data structures for tight coupling takes time $O(a \cdot p + m \cdot a)$. Computing the auxiliary data structures for histograms takes time $O(a \cdot p^2 + m \cdot a)$. Thus, the total time for selection is $O(a \cdot (r + m + p^2)) = O(a \cdot (r + p^2))$.

There are three components to the time for querying. Tight coupling takes time $O(a)$. Computing histograms takes time $O(a \cdot p)$. Updating the starfield takes time $O(u)$. Thus the total time for querying is $O(a \cdot p + u)$.

5. Experiments

Preliminary experiments show that the incremental approach can deal with an active subset consisting of 100,000 records with 10 attributes each [13]. In comparison, the pioneering work in the area, the Film Finder program [1], could handle a database of 10,000 records with 10 attributes, and some of the standard data structures analyzed in [10] and tested in [11] demonstrated scalability up to 20,000 records with 10 attributes.

The following subsections describe the experimentation on an implementation made by using our methods and show the results. First, we describe the implementations and the environment for the experiments. The experimentation method and the results are presented next. After this we show the derivation of the experimental run time behavior (complexities) obtained from the experiments. Then we test the validity of these experimental complexities and finally state some conclusions.

5.1. Experimentation Environment

We implemented a sample DQI using range sliders. The interface consisted of a starfield display, a preview bar (to show aggregate information about the query that is being formed), and a number of range sliders depending on the number of attributes in the input dataset. The starfield display and the points in the starfield display could have variable sizes. Also the range slider sizes could vary.

We used a SUN SPARC Station 5 with 32MB of RAM that runs a standard UNIX operating system for our experiments. Motif and C were used in our implementations.

We timed the setup, selection, and querying separately by considering CPU time spent for each operation (to avoid defects that might come from a multiuser environment). File read, data structure setup, and similar sub-setup and sub-selection times were also measured. Also all the experiments were repeated without a preview bar and a starfield display to measure the querying time (without giving any visual output to the user, this is the "pure" querying time).

We varied the total number of attributes, the total number of records, the starfield size, the point sizes on the starfield, the range slider sizes, and the jump sizes (the displacements, in pixels, of a single side of a range slider between consecutive queries). The experiments were controlled by a batch process. This was necessary to get accurate timings using exact jump sizes. We generated random numbers according to a uniform distribution for our datasets in our experiments. This was also necessary because the starfield display has its slowest performance when there are many pixels to be updated on the display. To make a worst case analysis, we tried to reduce the number of overlaps on the display. Although this might eliminate some of the computations for updating the overlapping points, it increases the starfield update times which is more expensive than the count updates.

We considered the worst case that might occur in a single selection operation. This occurs when m is equal to r that might not occur very frequently in real applications. So our experiments show some over-estimated times for querying, selection, and setup. In real-life applications we expect to observe better performances from our experiments.

5.2. Experiments and Results

We ran 7200 experiments to assess the performance of our implementations. 3600 of them ran with the starfield display and the preview bar enabled and the other 3600 ran with them disabled. The following values were used in our experiments:

Number of attributes (a) 2, 4, 6, 8, or 10

Starfield size (f) 400^2, 500^2, or 600^2 pixels

Point size (d) 1^2, 3^2, 5^2, or 7^2 pixels

Range slider size (p) 150, 200, or 250 pixels

Dataset size (r)
$10,000$, $25,000$, $50,000$, $75,000$, or $100,000$ records

Jump size/range slider size (j/p)
1/50, 1/25, 1/10, or 1/5

In the first 3600 experiments we measured the time to update the internal data structures (without any user interface updates). In general we observed that the "pure" querying time is no more than 20 milliseconds (average of 10 milliseconds). This was negligible, with respect to the starfield display times obtained, especially when the number of records got larger (with a starfield display that does not use our methods, querying time becomes even less significant with respect to the update times for the starfield display). This suggests that the starfield update times must be optimized first for a faster DQI.

The second set of 3600 experiments were used to measure the querying times including the starfield display update times. The complexity analysis depends on these remaining 3600 experiments, which we present in the following subsections (Figure 5).

5.3. Experimental Complexity

Let s_e denote the estimated setup time. Let s_o denote the setup time observed from our experiments. In an ideal analysis we must always observe an equality between the two times. Obviously there are some errors in the experiments and in the formula itself (due to neglecting some low-order terms). We ran a *multiple linear regression* on our experiments where $|s_o - s_e|$ is minimized over all the experiments. This is equivalent to finding the best constants for $s_e = A + B \cdot r \cdot a$ formula (again over all the experiments) where A and B are the constants and $r \cdot a$ term is obtained from the theoretical complexities given in O notation in previous sections. The setup has a constant A as we need to represent file open and close times spent in the experiments. After the regression we obtained $A = 1.16$ and $B = 0.0000177$. Hence, $s_e = 1.16 + 0.0000177 \cdot r \cdot a$.

We did a similar analysis with the selection times. Let S_e denote the estimated selection time. Let S_o denote the selection time observed from our experiments. Therefore, we can get the formula $S_e = B \cdot r \cdot a + C \cdot a \cdot p^2$. Selection does not have a constant like A as we had in setup so $A = 0$. The regression produced $B = 0.00000121$ and $C = 0.00000116$. Hence, $S_e = 0.00000121 \cdot r \cdot a + 0.00000116 \cdot a \cdot p^2$.

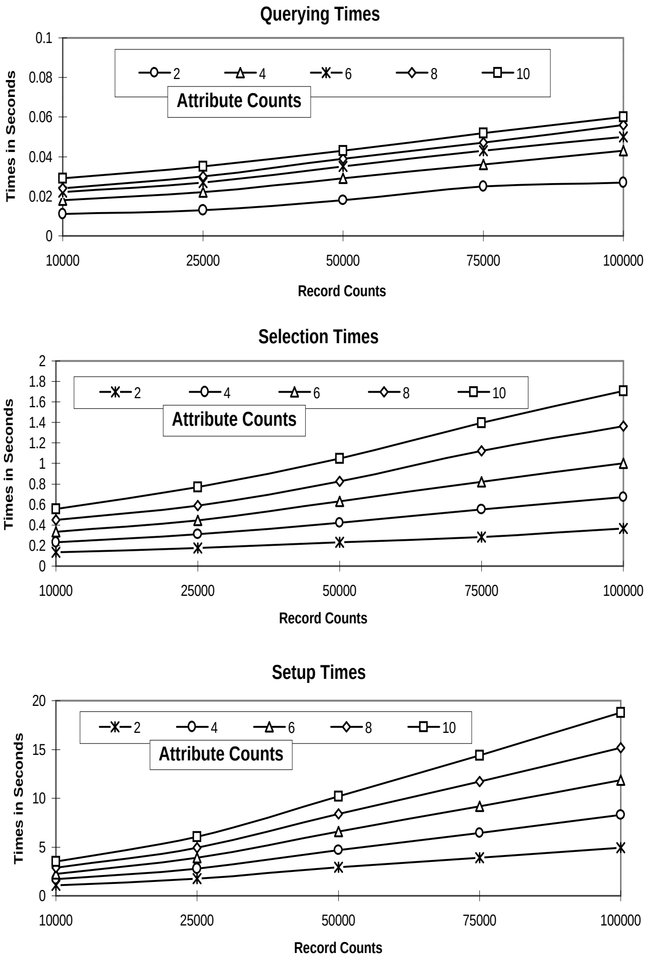


Figure 5. A subset of experiments (25 experiments out of 3600): The starfield size is 500^2 pixels, the range slider size is 200 pixels, the point size is 5^2 pixels, and the jump size is $200 \times 1/25 = 8$ pixels (which forms the average case for our experiments). The querying times given in this graph show that the average case for our experiments is an order of magnitude faster than the ones observed in the previous experiments with other data structures and algorithms.

Finally, let Q_e denote the estimated querying time. Let Q_o denote the querying time observed from our experiments. Again using the theoretical complexities we can get $Q_e = A + B \cdot a \cdot p + Cu$. Due to initialization routines, we found the existence of A appropriate in this formula (eventually it turned out to be a small value). Unfortunately, the analysis for querying is not trivial, as we have to find an estimate for u in terms of display size, point size, and etc. We saw that the u term is directly proportional to the jump size and the number of records needed to be updated on the starfield. Since we paint more than one pixel per point (in general) the formula counts the number of pixels that are updated. So our estimate for u is $r \cdot j \cdot d^2/p$ where d^2 is the number of pixels to be painted per each point and j is the jump size. Hence, $Q_e = A + B \cdot a \cdot p + Cu$ becomes $Q_e = A + B \cdot a \cdot p + C \cdot r \cdot j \cdot d^2/p$ for our case. Similarly, the regression produced $A = 0.00528$, $B = 0.0000157$, and $C = 0.000000263$. Hence, $Q_e = 0.00528 + 0.0000157 \cdot a \cdot p + 0.000000263 \cdot r \cdot j \cdot d^2/p$.

5.4. Evaluation

To evaluate the approach we used two methods. The first one is to run the X^2 test to assess the correlation between our experiments and the theoretical terms. Then we ran another set of experiments to see whether we can estimate the outcomes of these new experiments with our old formulas (and hence with the old constants) or not.

The X^2 test, for all of the three measurements (setup, querying, selection), showed that the estimated values and the actual values obtained from the experiments were highly correlated.

We ran 1000 new experiments and obtained the same times with the similar methods used for the previous set of experiments (the starfield display was always active in this set of experiments). We had a random combination of the following values for our experiments: point size varied between 1^2 to 10^2, jump size varied from 1 to 50, display size varied from 300^2 to 600^2 (with range sliders of size 250 pixels a user interface with a starfield of 600^2 pixels nearly fills the display), and the slider sizes varied from 150 to 250. The only values that were fixed during these new set of experiments (i.e., same values with the previous set of experiments) were the dataset sizes and the attribute counts (as it is practically impossible to generate all the possible (random) datasets (either in terms of time or space) on the fly for these new set of experiments). The differences between the estimates and the actual times were obtained. The average deviation observed for setup time was 9.50 percent; for selection, 3.97 percent; for querying, 16.63 percent.

5.5. Discussion

Using the incremental approach we achieved better querying times than previous implementations (that had standard data structures for querying which were not specifically designed for DQIs). We also consumed less memory as we created the data structures whenever they were needed. The new approach enabled us to give preview bar, histogram, and tight coupling information to the user without making any additional queries or spending additional processing times. The size of the main memory that is being used by the implementations remained as a secondary problem. We saw that there are problems in the selection times before we reached to the memory limits of our architecture (more than 1 second generally annoys the user). The selection times were mostly around 1 second. Hence, memory still remained as a secondary problem for DQIs. As r increases, terms that contain the r factor become more significant. The starfield display times were significant for huge r's and our approach gains its power from the incremental starfield display updates (but huge jumps in range sliders can still cause high display update times).

The average deviation for the selection times was smaller than we expected for the random set of experiments. The setup times were also acceptable as the disk input caused fluctuations in setup times. The querying time estimates were less accurate than we expected but were again acceptable. The reason for this was the high precision measurements that were made on a system that has lower precision settings than the required ones.

6. Conclusions

The new incremental approach for queries and display updates introduces a better way of dealing with large databases. Experiments show that this approach is faster than previous approaches and can deal with an order of magnitude of larger datasets (i.e., $100,000$ records with 10 attributes). The querying time is dominated by the starfield update time (also observed in [11, 13]). The incremental approach enables faster display because only the difference between consecutive queries is updated in the data structures and on the starfield display.

7. Future Work

Our goal is to make another order of magnitude increase in the size of the datasets that DQIs can deal with ($1,000,000$ records with 10 attributes). We plan to:

- implement other widget types, e.g., alphanumeric sliders.
- try spatial data structures like k-D trees to see how they effect the times for selection and querying. (As a general non-worst-case rule of thumb, spatial data structures answer range queries in time $O(|H|^{1-1/a})$ where H is the set of hits and a is the number of attributes in the input dataset. This could be good for selection, because it is sublinear. But it could be bad for querying, because it is close to linear, and prior work seems to confirm this doubt [10, 11, 13]. Instead, we will use an incremental approach where we compute the difference ΔH between consecutive hit sets, which in practice should take time only $O(|\Delta H|^{1-1/a})$.)
- combine our DQIs with a query previewer [3] in order to produce a new state of the art in interactive dynamic database access.

References

[1] Ahlberg, C. and Shneiderman, B., Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays, *Proc. ACM SIGCHI '94*, 1994, pp. 313–317.

[2] Ahlberg, C. and Wistrand, E., IVEE: An Information Visualization and Exploration Environment, *Proc. IEEE Information Visualization*, 1995, pp. 66–73.

[3] Doan, K., Plaisant, C., and Shneiderman, B., Query Previews in Networked Information Systems, *Proc. Forum on Advances in Digital Libraries*, IEEE Computer Society Press, 1996, pp. 120–129.

[4] Eick, S., Data Visualization Sliders, *Proc. ACM User Interface Software and Technology*, 1994, pp. 119–120.

[5] Fishkin, K. and Stone, M. C., Enhanced Dynamic Queries via Movable Filters, *Proc. ACM SIGCHI '95*, 1995, pp. 415–420.

[6] Goldstein, J. and Roth, S. F., Using Aggregation and Dynamic Queries for Exploring Large Data Sets, *Proc. ACM SIGCHI '94*, 1994, pp. 23-29.

[7] Human-Computer Interaction Laboratory (HCIL), University of Maryland at College Park, Homefinder, 1992. The down-loadable PC demonstration is available from ftp://ftp.cs.umd.edu/pub/hcil/Demos/DQ/dq-home.zip.

[8] Information Visualization and Exploration Environment (IVEE) Development AB, http://www.ivee.com/. Online Java demo and down-loadable demos for various platforms.

[9] Ioannidis, Y., Dynamic Information Visualization, *ACM SIGMOD Record*, Vol. 25, No. 4, 1996, pp. 16–20.

[10] Jain, V. and Shneiderman, B., Data Structures for Dynamic Queries: An Analytical and Experimental Evaluation, *Proc. Advanced Visual Interfaces*, Available from ACM, New York, 1994, pp. 1–11.

[11] Pointek, J., personal communication, reachable via email: pointek@cs.umd.edu, 1995.

[12] Shneiderman, B., Dynamic Queries for Visual Information Seeking, *IEEE Software*, Vol. 11, No. 6, 1994, pp. 70–77.

[13] Tanin, E., Beigel, R., and Shneiderman, B., Incremental Data Structures and Algorithms for Dynamic Query Interfaces, *ACM SIGMOD Record*, Vol. 25, No. 4, 1996, pp. 21–24.

[14] Williamson, C. and Shneiderman, B., The Dynamic HomeFinder: Evaluating Dynamic Queries in a Real-Estate Information Exploration System, *Proc. ACM SIGIR '92*, 1992, pp. 339–346.