Skimmer: Rapid Scrolling of Relational Query Results

Manish Singh
University of Michigan
Ann Arbor, USA
singhmk@umich.edu

Arnab Nandi
The Ohio State University
Columbus, USA
arnab@cse.osu.edu

H.V. Jagadish
University of Michigan
Ann Arbor, USA
jag@umich.edu

ABSTRACT

A relational database often yields a large set of tuples as the result of a query. Users browse this result set to find the information they require. If the result set is large, there may be many pages of data to browse. Since results comprise tuples of alphanumeric values that have few visual markers, it is hard to browse the data quickly, even if it is sorted.

In this paper, we describe the design of a system for browsing relational data by scrolling through it at a high speed. Rather than showing the user a fast-changing blur, the system presents the user with a small number of representative tuples. Representative tuples are selected to provide a "good impression" of the query result. We show that the information loss to the user is limited, even at high scrolling speeds, and that our algorithms can pick good representatives fast enough to provide for real-time, high-speed scrolling over large datasets.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: User Interfaces

General Terms

Algorithms, Design, Performance

Keywords

Fast browsing, Tuple sampling, Scrolling history

1. INTRODUCTION

Database query results often comprise hundreds, and even thousands, of tuples. Typically, these tuples are presented to a user through a scrolling interface. Since these tuples, typically, contain alphanumeric information and are devoid of visual cues, the user is easily overwhelmed. It becomes difficult for the user to perform meaningful tasks by scrolling through a large result set. The goal of this paper is to make the commonly used scrolling interface more usable for browsing large relational result sets.

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

It is often hard for users to specify precisely the query results of interest [8]. They are then faced with the problem of having an *empty answer* or *many answers* [2, 9]. *Empty answers* are returned when the query is too selective, whereas a query that is not selective enough returns *many answers.* Not surprisingly, user interface studies [14, 6, 8, 7] have shown that users tend to browse quite often while searching for information, in addition to querying. These studies show that browsing is a rich and fundamental part of human information seeking behavior, and that querying is not the only mode of interaction with data. Users often formulate a more precise query after they have attained an understanding of the underlying data through quick exploratory browsing.

Let us consider an example task to better appreciate our problem: *A realtor database that has a single table with attributes:* (`TID, Price, Bedrooms, Bathrooms, CrimeRate, Zipcode, ...`). *Each tuple represents a property for sale. Consider a user looking to buy a home who has a rough idea of the desired price, neighborhood and home size.*

In the example above, the user typically issues the query, and is then presented with a tabular collection of results, each tuple representing a property. While the users specifications were well articulated, even within the parameters specified, there are still several hundred homes. The user has to browse through all the result tuples to find a few homes of interest to see. This browsing is typically done by scrolling through the large table of results, a daunting task.

To help the user quickly get a sense of large result sets, various sophisticated data reduction techniques such as clustering and faceting, are available (see Section 6 for details). While there are situations in which such techniques can be effective, they also have significant limitations. First, clusters do not help the user unless they have labels (or other means) that clearly identify what can be found inside each cluster. Thus, a full scan is still required. For example, if the user were looking to buy a home, she would probably insist on browsing the result set rather than relying on some realtor's clustering software. Secondly, it is important to let the user control the rate of consumption of information. For tabular data, the user can slow down or speed up her reading speed by moving the scrollbar accordingly. In a clustering-based system, one would need to allow the user to dynamically vary the granularity of each cluster in an interactive fashion. Such a solution is not only computationally challenging but is also unintuitive for quick browsing.

Where user preferences are well understood, results may be scored and ranked. Information retrieval systems routinely present large result sets in this fashion. Users can

then focus on the few top-ranked results and skim, or even ignore, the rest. However, in the database context there is usually insufficient information to rank with confidence. A common solution in the case of databases is to sort the result set on some attribute of importance: say `Price`, in our realtor example. The user then has to scroll through this large result, reading through each tuple. While the data is sorted on the `Price` column, the user still has to focus on reading the other columns, which may not correlate with `Price`. Clearly, reading through each non-sort-key attribute of each tuple in a entire large result set is a slow and arduous task. Therefore, our task becomes that of supporting fast browsing (and more specifically, scrolling) over such a sorted relational result set.

Humans browse all the time, for example with newspapers and magazines. Visual cues in the layout assist users in browsing data quickly. Relational data tuples tend to comprise dense alphanumeric data, with few visual markers. Thus, there is a rather low maximum rate at which a human can skim tuples from a database. The only way to browse a database faster than this rate is to have the human eye see less than all the information.

To improve the usability of scrolling interfaces, we present in this paper a variable-speed scrolling interface that can automatically adjust the amount of information displayed based on a user's scrolling speed. Our interface displays only a few selected tuples from each page, where the number of tuples is determined by the user's current scrolling speed. If some results seem interesting to the user while scrolling fast, the user can reduce the speed of scrolling, making the system show more tuples from the currently-viewed section of the data. In contrast to a typical sampling problem, our goal is to identify tuples that provide the most information with respect to the *entire scrolling session*, and not just the page at hand.

While the specific sorting attribute is not material for our algorithms, the fact that data is sorted/clustered in some way is crucial. For fast browsing to work, we must have "similar" records close together, for some user notion of similarity. If we did not have this, then selecting any set of representative tuples would permit only global conclusions, but not local ones: there would be no reason to expect other interesting entries in the vicinity of an interesting observed sample. There are many ways interesting records could have been placed together—for example, a clustering algorithm could have been used. The algorithms in our paper will work in such a scenario as well. We chose to focus on sorting because this is so commonly used in database user interfaces.

Contributions: Our key contributions are as follows:

- **Scrolling-aware browsing:** We identify the problem of deriving representative tuples in the context of scrolling through sorted relational data.
- **Information loss metric:** We provide precise mathematical definitions that quantify loss of information incurred while browsing just the representative tuples.
- **Algorithms:** We develop and compare five new scrolling-based sampling algorithms that minimize information loss.
- **Interaction constraints:** Due to the interactive nature of our use-case, the efficiency of our algorithms is a key consideration. We provide efficiently computable algorithms that satisfy this fast scrolling requirement.

Paper Layout: The rest of the paper is organized as follows. In Section 2, we formally define our problem. We then describe the scrolling interface and define metrics for information loss. In Section 3, we introduce two naïve algorithms and propose five new algorithms to solve the problem at hand. We present experimental results in Section 4 and a user study in Section 5. Related work is discussed in Section 6, followed by conclusions in Section 7.

2. SCROLLING INTERFACE

In this section, we describe our formal problem set up, user interface, and metrics used to measure information quality.

2.1 Problem Definition

A user gives an `ORDER BY` SQL query $\mathcal{Q}$, which is executed on database $\mathcal{D}$ and it generates a result set $\mathcal{R}$. The query can be over a single table or joined over multiple tables. $\mathcal{R}$ contains N tuples and it requires S pages for display, i.e., pages $\{P_1, P_2, \cdots, P_S\}$. All pages, except the last page P_S, contain M tuples, where M is determined by the page and tuple size. For the remainder of the paper we use the page size to indicate the number of tuples per page (i.e., M).

Our preliminary user study experiments (not reported here) showed that for alphanumeric, relational data, it is easier for users to see information using intermittent page-wise scrolling, as compared to continuous scrolling. In intermittent page-wise scrolling, the display contents are refreshed only when the scrollbar moves to adjacent pages. We believe this is because it is harder for a user to read continuously moving tuples, as found in a continuous scrolling interfaces. Based on this design, in our proposed scrolling interface, a user goes through a much smaller result set $\{D_1, D_2, \cdots, D_S\}$, where D_i is the set of representative tuples shown from page, P_i.

Our system computes the set D_i, for page P_i, based on user's current browsing speed, i.e., in our implemented system it is the average browsing speed at page P_{i-2}. Due to computational constraints, the tuples are selected when the user is currently browsing through page P_{i-1}. We compute K_i, the size of D_i, based on a straightforward inverse function of scrolling speed, discussed in Section 5.4. Our central problem is to select D_i that gives the user a good impression (defined later in Section 2.3) of page P_i, and at the same time, avoids showing redundant information. Furthermore, this selection has to be performed very fast, so that the user is not hindered by our system even when browsing the data set at high speed.

2.2 User Interface

It should be noted that designing a user interface is not the primary aim of our work—rather, our goal is to design algorithms that can identify representative tuples that provide high quality information to the user within the real-time constraints of fast browsing.

A user can scroll through the result pages, $\{P_1, P_2, \cdots, P_S\}$, arbitrarily varying her scrolling speed. At a *slow speed*, all the tuples are shown; whereas in the *high speed* case, only a few selected tuples are shown. In Figure 1, we show a page of tuples containing properties from a realtor database. Through three tuples, our system is able to give a good overall impression of the full page. If the scroll speed changes and a page is redisplayed with more tuples, the relative location of the originally displayed tuples remains fixed, giving

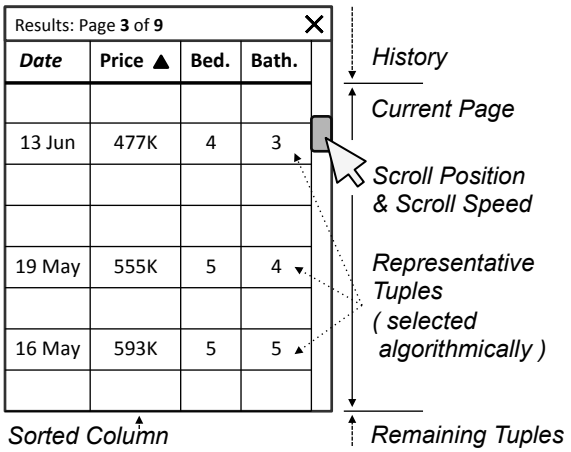

Figure 1: *User interface:* Results are displayed in a paginated interface, browsable using the scrollbar. The scroll position determines the currently displayed page. Instead of overwhelming the user with a full display where all tuples are shown, the user is presented with a condensed display featuring a set of representative tuples. These tuples are selected from the current page based on contents of the page itself, the history of pages seen so far, the scroll position and the speed of scrolling.

the user a point of reference and allowing her to read information that is stationary. In addition to the tuples, we show the page number of currently displayed page.

In our interface, when a user scrolls backwards (i.e. scrolling towards beginning of the result set), we display all the tuples (full display), but when a user scrolls in forward direction (scrolling towards the end of result set), we display only selected representative tuples. During the forward scroll, if the user's average scroll speed is below a threshold speed V_t, then all the tuples will be shown. Thus, a user can see all the tuples in page P_i, by either scrolling in forward direction below the threshold speed at page P_{i-2}, or scrolling in backward direction at page P_i. The reason to maintain this asymmetry is because users are expected to scroll forward most of the time, but we want to give them a way to stop suddenly when something catches their eye.

2.3 Goodness Measure

We address the issue of information overload encountered during fast browsing by displaying only a select set of representative tuples from each page, instead of showing all tuples that satisfy the user's query. In other words, we show the user less information, which brings up the question of information loss. Can we quantify the information we did not show the user?

Entropy-based information loss measures are well studied in the area of Information Theory. However, we were not able to use these measures for our specific problem because defining row wise entropy, or significance of an individual tuple with respect to the whole result set, is hard, since all values in a row are distinct. To the best of our knowledge, there is no existing work that mathematically quantifies this kind of information loss. In this section, we develop some natural metrics for this purpose. To do so, we first define two preliminary concepts: *scroll log* and *history*.

Scroll Log: A scroll log (SL) maintains the sequence in which pages of a query result were visited by a user. It logs three things: sequence number *sid*; page number *pid*; and a list of tuple ids *tupleList*, of all displayed tuples from page *pid*. Thus, the log has the form (*sid, pid, tupleList*), where sequence id *sid* is incremented with every new page visit during a forward scroll.

History: We call already displayed information as *history* and denote it by *H(sid)*. *H(sid)* contains the ids of all tuples that have been shown to the user, prior to scrolling in the *sid* page in sequence. The list of tuple ids shown from this page is denoted as *tupleList(sid)*. Thus $H(sid) = \bigcup_{i=1}^{sid-1} tupleList(i)$. Note that with a succession of forward and backward scrolls, the history of a page P_i could contain information from pages after page P_i.

We now go on to define our metrics for information loss in the context of browsing data: *Tuplewise Information Loss*, *Pagewise Information Loss* and *Cumulative Information Loss*.

DEFINITION 1. *The **Tuplewise Information Loss (TIL)** score of a non-displayed tuple t_{nd}, from page P_i having sequence id 'sid' in scroll log, is the dissimilarity of t_{nd} with respect to the most similar tuple t_d, from history $H(sid)$ and tuples shown from page P_i. Thus,*

$$TIL(t_{nd},\ sid) = \mathcal{V}(t_{nd},\ t_d) \tag{1}$$

Here, $\mathcal{V}$ gives the dissimilarity between t_{nd} and t_d, where $t_d \in H(sid) \cup tupleList(sid)$.

We can use any distance function $\mathcal{V}$ to measure dissimilarity. In the experiments reported in this paper, we use simple Euclidian distance after scaling dimensions to comparable ranges. Intuitively, $TIL(t_{nd},\ sid)$ gives a measure of the most similar information with respect to t_{nd} that has been shown to the user from either already displayed tuples $H(sid)$ or tuples that has been shown from page P_i, *tupleList(sid)*.

DEFINITION 2. *The **Pagewise Information Loss (PIL)** score for a page P_i, is defined as the sum of TIL scores of all tuples from P_i. For pages which are not scrolled or visited, PIL score is zero. Thus,*

$$PIL(P_i,\ sid) = \sum_{t_p \in P_i} TIL(t_p,\ sid) \tag{2}$$

Since the *TIL* score of all displayed tuples is zero, the *PIL* score of a page is the sum of the *TIL* score of all non-displayed tuples.

To minimize information loss even during fast scrolling, we need to ensure that we do not show redundant information, i.e., information that is similar to what has already been shown. For example, in our realtor application, if a user has sorted the houses by the `Price` attribute, then it is not desirable to show expensive houses that are very similar to cheaper houses that have already been shown to the user in previous pages. A buyer is willing to look for expensive houses only if the expensive houses have features which are significantly different from lower priced houses. To avoid redundancy, while selecting tuples from any page P_i, we need

to take into consideration the information that has been shown prior to display of P_i, which we maintain in the form of history.

The relationship between history and *PIL* score can be understood more clearly through Figure 2, where we have shown the selected tuples (current representatives) from two types of sampling algorithms—one that considers no history (local sampling) vs one that consider displayed history (history based sampling). All the circles represent tuples in a page of a binary relation, and we have to select two representative tuples from this page. Clearly by selecting the two representative tuples as shown in Figure 2 (a), we reduce the overall *PIL* score for the page. The problem with local sampling can be understood by seeing the already displayed tuples (i.e., historical representative) in Figure 2 (b). The two selected tuples in local sampling do not provide as much additional information as compared to what the user had already seen in the past. As shown in 2 (b), considering history enables us to provide new information that is substantially different from previously shown information.

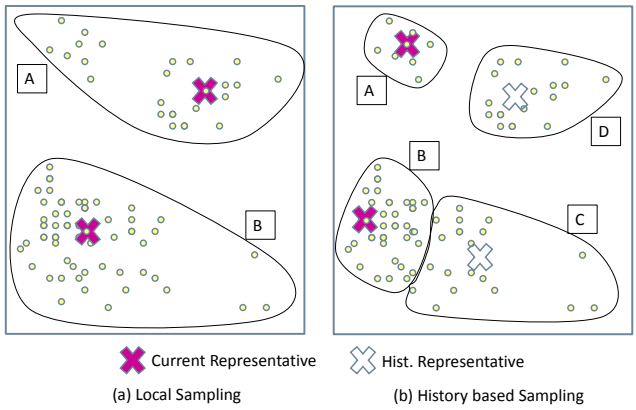

Figure 2: Local and Historical Sampling

DEFINITION 3. *The **Cumulative Information Loss (CIL)** score for a scroll log 'SL' and result set $\mathcal{R}$, is defined as the sum of PIL score of all entries in 'SL'. Thus,*

$$CIL(SL, \mathcal{R}) = \sum_{sid=1}^{|SL|} PIL(pid,\ sid) \tag{3}$$

Thus, *CIL* score is the sum of *PIL* scores of all entries in scroll log *SL*. Intuitively, given two sampling algorithms A and B, one can say that algorithm A is better than B in terms of minimizing information loss, if $CIL_A(SL, \mathcal{R}) < CIL_B(SL, \mathcal{R})$, where $CIL_X(SL, \mathcal{R})$ denotes the cumulative information loss incurred by sampling according to algorithm X, given the result set $\mathcal{R}$ and scroll log *SL*.

3. ALGORITHMS

In this section, we present two naïve and five novel sampling algorithms that minimizes *CIL* score. Since *CIL* score is sum of pagewise *PIL* scores, these algorithms minimize individual pagewise *PIL* score. For the sake of simplicity, we assume that we have a data page P, with associated history H, and that we have to select K representatives from page P. The collection of selected representatives is called the display set D. The sizes of the page P and history H are denoted by M and $|H|$, respectively. Literature in the clustering domain generally use the terminology *object*, which is equivalent to our *tuples*. We use these two terms interchangeably when describing our sampling algorithms.

3.1 Naïve Sampling Techniques

To the best our knowledge, there is no existing work that supports variable-speed scrolling for relational data, and thus we use two sampling techniques, namely random and uniform sampling, as our naïve solutions to evaluate and compare against. In these naïve solutions, we pick K random or uniformly spaced tuples from page P. We refer to these naïve sampling techniques as RS and US, respectively. Since the data is sorted, we expected uniform sampling to give a good overall impression because it will select tuples at regular interval.

3.2 K-medoids based Sampling Techniques

3.2.1 Relationship between PIL Score and K-medoids

Minimizing the *PIL* score (i.e., Definition 2) is equivalent to solving the standard K-medoids clustering algorithm, with some modifications. K-medoids is a partitioning based clustering algorithm that divides a given set of objects into K partitions and returns an actual representative object from each partition or cluster.

In K-medoids, we try to minimize the following absolute error criterion:

$$E_{K\text{-medoids}}(P) = \sum_{j=1}^{K} \sum_{p \in C_j} \mathcal{V}(p, o_j) \tag{4}$$

Here, $E_{K\text{-medoids}}$ is the sum of absolute error for all objects in the dataset P; p is an object in cluster C_j, which is represented by object o_j. $\mathcal{V}(p, o_j)$ measures the error in representing object p by object o_j. $\mathcal{V}$ could be any generic dissimilarity function, including functions over objects having categorical features.

Minimizing the *PIL* score of page P is equivalent to minimizing the following equation:

$$PIL(P) = \sum_{j=1}^{L} \sum_{p \in C_j} \mathcal{V}(p, o_j) \tag{5}$$

where, $L = |H| + K$. While minimizing Equation 5, we can only select K new representatives from page P, the $|H|$ representative objects shown in previous pages cannot be changed.

The mathematical formulation for K-medoids and *PIL* score looks quite similar, with the difference that in the *PIL* score, we have fixed cluster centers from previous pages and that we want to avoid picking new centers which are close to the already displayed cluster centers, as discussed earlier in Section 2.3.

3.2.2 Local K-medoids

In the Local K-medoids (*LKMed*) sampling algorithm, we do not consider the effect of history while minimizing Equation 5. We compute the representatives using the normal K-medoids formulation, i.e., Equation 4. Any standard K-medoids algorithm can be used for selecting the K representatives from page P, we use PAM (Partition Around Medoids) [17]. For large datasets, more efficient algorithms

such as CLARA [17] and CLARANS [25] are generally used, which use sampling and randomization, respectively, to reduce the computational cost of PAM. To get reasonable results, these algorithms still need at least several dozen points after the sampling. Since we are clustering one page of tuples at a time, we only have a few dozen tuples to cluster. For these reasons, the extra machinery to handle large data sets is not required and PAM is our preferred algorithm.

PAM is an iterative clustering algorithm that requires $O(K.(M-K)^2)$ distance computations per iteration, where M is dataset size. For typical values of M (say 50) and K (say 10) in our application, this is still several thousand computations that must be performed within time so short (milliseconds) that the interface feels responsive to the user. Since M is small enough that we can afford M^2 storage in memory, we improved the performance of PAM by computing the distance between all tuples of a page once, and then using it in all iterations. This avoids the need to repeatedly compute distances between tuples, as required in classic PAM [17].

3.2.3 Historical K-medoids

Historical K-medoid (*HKMed*) modifies *LKMed* by keeping history H and modifying the PAM algorithm in light of H. Samples in *HKMed* are computed according to Algorithm 1. While *HKMed* seems to have additional computation steps as compared to *LKMed* because it takes history into account, both our theoretical and experimental evaluation show that *HKMed* is better than *LKMed*, both in terms of information quality and computation time (a more detailed explanation is provided in the forthcoming paragraphs).

Algorithm 1 Historical K-medoids

Input: K: size of display set
P: a data page
H: displayed history
Output: D: display set from page P
Method:

1: $flag \leftarrow true$
2: map $NH \leftarrow NearestNeighbor(P,\ H)$
3: map $TD \leftarrow AllPairDistance(P)$
4: $D \leftarrow K$ random objects from P as initial representatives
5: **repeat**
6: map $NM \leftarrow NearestNeighbor(P \setminus D,\ D)$
7: map $NC \leftarrow MergeNearestNeighbor(NH,\ NM)$
8: for all pairs of objects o_m, o_p, where $o_m \in D$ and $o_p \in P \setminus D$, compute $TC_{mp} \leftarrow SwapCost(o_m,\ o_p)$
9: **if** $(Min(TC_{mp}) < 0)$ **then**
10: $D \leftarrow (D \setminus o_m) \cup \{o_p\}$
11: **else**
12: $flag \leftarrow false$
13: **end if**
14: **until** (flag = true)
15: return D

Algorithm 1 differs from the basic PAM algorithm in the addition of steps 2 and 7. In basic PAM, we perform step 4 to select the initial representatives and then keep repeating steps 6, 8-14 till convergence. In the above algorithm, '\' refers to the set difference operator. In step 6, we compute the nearest neighbor of all non-displayed objects (tuples) from page P with respect to the currently selected representatives D. In steps 8-14, the goal is to swap one of the currently selected representative objects o_m with a non-selected object o_p that gives the greatest reduction in error cost. The best pair is one for which the TC_{mp} value is minimum and this value should be negative. We would refer the readers to [17, 25] for more details of the basic PAM algorithm. For both *LKMed* and *HKMed* algorithm, each iteration involves computing *SwapCost* $K.(M-K)$ times. In the *SwapCost* algorithm, we need to consider the effect of swap on all the $(M-K)$ non-selected objects, and thus the overall complexity is $O(K.(M-K)^2)$.

In step 2, we compute the nearest neighbor for all tuples in page P with respect to history H. We keep both tuple id and distance in the map NH. In step 4, to avoid repeated distance computation between tuples in P, we pre-compute and store the distance of all tuples from each other in the map TD. Step 4 is useful even in basic PAM, as it can reduce the cost for computing the nearest neighbor in step 6, and second nearest neighbor in step 8, i.e., *SwapCost* algorithm. In step 7, we merge the nearest neighbor from historically displayed tuples H and the currently selected representative tuples D. The *SwapCost* algorithm has the same four cases as that of basic PAM algorithm [17, 25], with the exception that only the representatives from D (and not H) can be swapped with non-selected representatives from $P \setminus D$. The swap costs can be efficiently computed by using the information in map NC of step 7 and map TD of step 4.

As we will see experimentally in Section 4.2, *HKMed* is better than *LKMed* both in terms of information quality and computation time. The better information quality is because *HKMed* exactly minimizes the *PIL* score (*LKMed* returns samples like Figure 2 (a), whereas *HKMed* returns samples like Figure 2 (b)). While *HKMed* seems to have two additional steps, i.e., step 2 and 7, it is computationally faster than *LKMed*. *HKMed* is faster due to reduced computation in *SwapCost* algorithm of step 8. In the *SwapCost* algorithm, there are four cases to consider for each non-selected object o_j, when we replace an existing medoid o_m by a new medoid o_p (see details in [17, 25]). For a non-selected object (o_j) that is in o_m's cluster, we have to consider cases 1 and 2, where we need to compute the second nearest neighbor from D to compute the replacement cost. In case o_j is not in o_m's cluster, we need to consider cases 3 and 4, which are faster to compute. In case of *HKMed*, we have a number of fixed medoids from history H and thus the number of times we need to consider cases 1 and 2 for the selected medoids from P (i.e., D) would be many fewer than for *LKMed*.

3.3 K-means Based Sampling

Each iteration of K-medoids based clustering algorithms requires $O(K.(M-K)^2)$ distance computations, and thus may not be suitable for very fast scrolling. The cost of K-medoids based algorithms increase with an increase in page size or sampling rate. To address these computational constraints, in this subsection, we present three K-means approximated to K-medoids based sampling algorithms, which are computationally much faster, with $O(K.M)$ distance computations per iteration. One of these algorithms return representatives that are almost as good as *LKMed*.

K-means is also a partition based clustering algorithm, but it returns the mean of the objects in a cluster as clus-

ter representative, unlike K-medoids that returns an actual object from the cluster. A cluster center obtained from K-means may not be a real object in the cluster. To get a K-medoids approximation, we return K-actual objects that are nearest to each of these cluster centers. K-means is restricted to Euclidean distance functions, whereas K-medoids can support any distance function $\mathcal{V}$. K-means minimizes the following square-error criterion:

$$E_{K\text{-means}}(P) = \sum_{j=1}^{K} \sum_{p \in C_j} |p - m_i|^2 \quad (6)$$

Here, $E_{K\text{-means}}$ is the sum of square error of all objects in the dataset P; p is an object assigned to cluster C_j, and C_j is represented by m_j, i.e., mean of all the points in C_j.

3.3.1 Local K-means

Local K-means (*LKMeans*) is similar to *LKMed*, where we do not consider the effect of history H. We compute K cluster centers from page P using the basic K-means clustering algorithm [24]. For each cluster center, we compute the nearest neighbor from P and these tuples constitute the display set D. This algorithm takes $O(K.M)$ distance computations per iterations and is suitable for very fast scrolling. Note that for large datasets, more efficient K-means algorithms are available, such as grid based optimizations proposed in [21]. However, in our problem, since page-wise data is small, basic K-means would be quite fast and give the best quality result.

LKMeans has the same disadvantage as *LKMed*, discussed in Section 2.3 Figure 2 (a); that it may display tuples which are very similar to already shown tuples. To avoid redundancy, we now present two K-means based algorithm that take history in account.

3.3.2 Historical K-means

For Historical K-means (*HKMeans*), see Algorithm 2, we make similar modifications to *LKMeans*, as we did from *LKMed* to *HKMed*.

Algorithm 2 Historical K-means

Input: K: size of display set
P: a data page
H: displayed history
Output: D: display set from page P
Method:

1: $D \leftarrow K$ random tuples from P as initial cluster centers
2: map $NH \leftarrow NearestNeighbor(P, H)$
3: **repeat**
4: re(assign) each tuple in P to its closest cluster center in $D \cup H$
5: **for all** cluster centers d in D **do**
6: if d has empty assignments, split the largest cluster assignment in D into two equal parts and assign half of the tuples to d
7: **end for**
8: update cluster centers in D to the mean of its constituent instances
9: **until** no change
10: $D \leftarrow NearestNeighbor(D, P)$
11: return D

Algorithm 2 is similar to basic K-means clustering algorithm [24], with a difference in steps 2 and 4, where we need to consider the effect of history H. In step 4, we assign each tuple in P to the nearest cluster center in $D \cup H$. If computed, as in case of normal K-means, step 4 would require $O((|H| + K).M)$ distance computation per iteration. However, this can be computed more efficiently by using the nearest neighbor map NH, computed in step 2. In step 4, we should compute the nearest neighbors of P with respect to D only, and then see if these nearest neighbors are closer than the nearest neighbor from history H (i.e., map NH). Since the history is fixed, we do not need to repeatedly compute the nearest neighbor with respect to H. This efficient computation will require $O(K.M)$ distance computations per iteration in step 4, which is same as basic K-means. For step 2, we would need an additional $O(|H|.M)$ distance computations for computing the nearest neighbors of P with respect to H. In step 8, we update cluster centers in D, by taking the mean of all the tuples assigned to each cluster center in D. In step 10, to get a K-medoids approximation, we return K-actual tuples from P which are nearest to each of the computed cluster centers in D.

To make the *HKMeans* algorithm terminate with desired number of representatives, we need to address the empty cluster assignment problem. In *HKMeans*, tuples whose nearest neighbor from history H is closer than the nearest neighbor in currently selected representatives D, do not play any role in determining the new cluster centers D for next iteration. Since the page size is generally small and history may be large, the cluster assignments may often become empty, causing the algorithm not to converge or to terminate with desired number of representative tuples. In order to solve this empty assignment problem, in steps 5-7, if there is any empty cluster in D, then we split the largest cluster in D into two equal parts and assign half of the tuples to the empty cluster.

3.3.3 Two Phase K-means

In our empirical experiments, we were expecting *HKMeans* to yield a lower *CIL* score as compared to *LKMeans*, but to our surprise, the *CIL* score from *HKMeans* was quite high; in fact it was quite close to random sampling. By observing individual page-wise *PIL* scores, we observed that even though for most pages *LKMeans* and *HKMeans* were comparable in terms of *PIL* score, there were pages where the *PIL* score of *HKMeans* was quite high as compared to *LKMeans*. This apparently high *PIL* score was due to selection of outliers as representative tuples, as shown in Figure 3 (a).

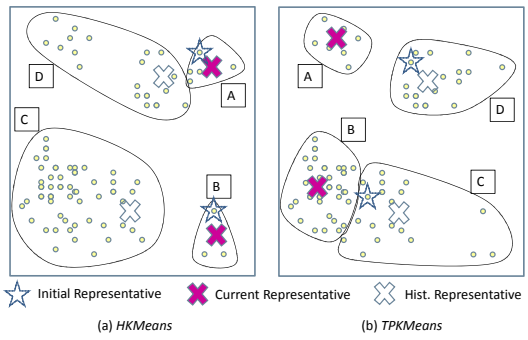

Figure 3: Local and Historical Sampling

Figure 3 (a) shows the effect of initialization (step 1) on Algorithm 2. In *HKMeans* tuples whose nearest neighbor from history H are closer than the nearest neighbor in currently selected representatives D do not play any role in determining the cluster center for next iteration. As shown in Figure 3 (a), if we pick the initial representatives from outliers (i.e., cluster A and B in Figure 3 (a)) and have historical representatives as shown, then since most of the tuples are closer to historical representatives, Algorithm 2 will converge with current representatives as mean of cluster A and B. Clearly, the selected representatives from B do not give a good overall impression as it is a cluster of outliers, and the representative from A is very similar to already displayed center from D. Unlike *HKMed* where the algorithm selects the medoids by computing the best possible swap in each iteration, *HKMeans* does cluster assignment simply based on nearest neighbor and thus may end up in a bad local optimum solution. Since the fixed centers from the history are not updated, and only a few tuples may play a role in selecting the new cluster center, *HKMeans* returns bad representatives quite often. This effect of initialization increases with increase in size of history (i.e., high sampling rate or size of truncated history L).

To address this problem of wrong initial cluster centers, we present a Two Phase K-means (*TPKMeans*) algorithm, which uses *LKMeans* in the first phase to compute proper initial cluster centers, and then in the second phase uses *HKMeans* algorithm to select final representative tuples. Figure 3 (b) shows that if we select proper initial cluster centers and then run Algorithm 2 using these as initial cluster center, we can avoid selecting outliers as representatives and also avoid showing redundant information.

TPKMeans is expected to perform quite well, as is confirmed through our experiments, because it initially chooses the centers from most dense locations and then moves to those portions of the data where there is no historical representative. The information quality from *TPKMeans* is quite close to *HKMed*. Computational complexity of *LKMeans* is $O(t.K.M)$, *HKMeans* is $O(t.K.M + |H|.M)$ and *TPKMeans* is also $O(t.K.M + |H|.M)$. Here t indicates the number of iterations.

3.4 Limiting the History

Using the entire history to minimize information loss, as discussed in Section 2.3, may not be practical from a system design perspective. This is because the size of history goes on increasing as the user continues to scroll. Ultimately, the history could become as large as the entire result set. Therefore, we consider using a truncated history that contains only information that a user has seen recently.

The truncated history can be computed by using the scroll log SL. For a page visit with sequence id *sid*, we scan SL backwards starting from *sid* and take the union of all *tupleList*s from the first L distinct pages that appear in a backward scan. If a page appears multiple times while looking for L distinct pages, we take the *tupleList* corresponding to the most recent visit of that page in SL. Of course, in a running system we do not need to go through this exercise repeatedly—it is straightforward to incrementally maintain the truncated history for the most recent L page visits, dropping the oldest one when a new page is visited.

The size of this truncated history, L, is a system parameter. We show experimentally that, as long as the value of L is not too small, having a large L does not lead to lower information loss. Thus, we can truncate history without hurting the quality of our results.

4. EXPERIMENTS

In this section, we report on the experimental evaluation of our variable-speed scrolling interface. We implemented the user interface and all sampling algorithms as described in Section 2 and Section 3, respectively. We performed experiments to compare the performance of all seven sampling algorithms in terms of their computation time and information quality. We also conducted a user study to measure the usability of a traditional full display vs. our sampling-based variable-speed scrolling interface, which we report in the next section.

4.1 Experimental Setup

We implemented our system in Java using the MySQL 5.5 RDBMS. Experiments were run on a dual-core Pentium 2.5 GHz PC with 2 GB of RAM. We used three datasets—Stock [18], Abalone [13] and ImageSegmentation [13]—for our user studies.

The Stock dataset has 2150 tuples, where each tuple represents stock details of a company for a day during year 1994-2003. It has 6 attributes, such as `Date`, `Starting Price`, `Max Price`, `Min Price`, etc. The Abalone dataset is generally used for regression analysis, where one predicts the age (age in years = number of rings + 1.5) of abalone from physical measurements, such as, `length`, `diameter`, `height` and different types of weights. This dataset has 4177 tuples with 8 attributes. ImageSegmentation is generally used for classification tasks. It has 2310 tuples from seven outdoor images: grass, path, window, cement, foliage, sky and brickface, where each tuple corresponds to a 3x3 region with 19 attributes. Each of the seven types of images has 330 instances.

4.2 Performance

In this subsection, we present the performance comparisons—both in terms of computation time and information quality—of all seven sampling algorithms on the Abalone dataset. While we conducted performance experiments on all datasets with similar results, we present our performance only on one dataset due to space constraints.

For our experiments, we assume a simple scrolling motion: a one-pass, constant speed, forward scrolling action from first to last page, with each page displaying a fixed number of tuples.

For these experiments, we assume the whole Abalone dataset as a sample query result. We present the effect of parameters such as page size, number of dimensions, sampling rate on the performance of these algorithms. Each of these plots are based on average readings of 25 simulation, where for each simulation we generate a different query result by selecting different random combination of attributes and ordering it by one randomly chosen attribute. The default parameters in these experiments are *page size* = 40 tuples per page, *number of dimensions* = 4 and *sampling rate* = 5 tuples per page. We have kept the size of truncated history, $L = 10$.

4.2.1 Computational Comparison

Figures 4, 5 and 6, measure the computation time of all the seven sampling algorithms with varying page size, number of dimension and sampling rate, respectively. We plot the average time for selecting tuples from each page, which is equal to the total time for a single pass simulation divided by the total number of pages. The average computation time gives a measure of the maximum allowed scrolling speed that a particular sampling algorithm can support.

Trends in Figures 4, 5 and 6 show that K-medoids based sampling algorithms take considerably more time as compared to the other five sampling algorithms. As discussed earlier, the computational complexity of *LKMed*, *HKMed*, *LKMeans* and *HKMeans* are $O(t.K.(M-K)^2)$, $O(t.K.(M-K)^2+|H|.M)$, $O(t.K.M)$ and $O(t.K.M+|H|.M)$, respectively. Here, t is number of iterations and $O(|H|.M)$ is the complexity of computing nearest neighbors with respect to history. *HKMeans* and *TPKMeans* have the same computational complexity. All three graphs show that the time taken by *LKMed* is greater than *HKMed*, even though in terms of computational complexity *LKMed* seems to be better than *HKMed*. As we had discussed earlier in Section 3.2.3, *HKMed* is faster than *LKMed* due to reduced amortized cost of *SwapCost* algorithm, in step 8 of Algorithm 1. *HKMeans* takes more time as compared to *LKMeans* because *HKMeans* requires additional computation for computing nearest neighbor in step 2 and during cluster assignment in step 4 of Algorithm 2. *TPKMeans* has slightly more computation time as compared to *HKMeans* because of the additional run of *LKMeans* needed in *TPKMeans* to get the initial cluster centers.

Figures 4 and 6 show that K-medoids based sampling algorithms cannot support fast scrolling for large page sizes or high sampling rates. K-means based sampling algorithms are quite fast even for reasonably large page sizes and high sampling rates. We will see later in Section 4.2.2 that the information quality obtained from *TPKMeans* is quite close to the K-medoids based sampling algorithms. Further, we can conclude from the three computation graphs that *TPKMeans* is suitable for very fast scrolling in all practical range of parameters.

Figure 6 shows that for both *HKMeans* and *TPKMeans* sampling algorithms, the computation time increases rapidly with an increase in sampling rate. This increase is due to large size of history (truncated history) for which these algorithms need to compute the nearest neighbors. Moreover, in the presence of a large number of previous fixed centers, these algorithms require more iterations to converge. When the history is large, the cluster assignments do not converge quickly because in each iteration, only a few tuples, which are not near to any of the historical centers, are used to compute the next step's cluster centers.

4.2.2 Information Quality Comparison

The metric for the quality of information result presented is the *Cumulative Information Loss (CIL)* defined in Section 2. As we change the problem set up parameters, such as page size or number of dimensions, the value of *CIL* changes greatly. Furthermore, *CIL* is not scaled to anything—there is no way to tell whether a given absolute value is good or bad. What we can say, however, is that smaller values are better. For these reasons, we define the notion of information gain below. The information gain is what we plot in our figures.

DEFINITION 4. *The* ***Information Gain*** *of an Algorithm A with respect to another Algorithm B is defined as:*

$$IG(A,B) = \frac{CIL_B(SL,\mathcal{R})}{CIL_A(SL,\mathcal{R})} \tag{7}$$

Figures 7, 8 and 9, show the information gain of each of the seven sampling algorithms with respect to random sampling, i.e., $IG(X, RS)$, where X represents a sampling algorithm. These graphs are obtained by varying page size, number of dimension and sampling rate, respectively. All the three graphs show that *HKMed* gives the highest information gain followed by *TPKMeans* and *LKMed*. In all the graphs, we can also see that the information gain from *HKMeans* is quite low and is very close to random sampling.

The large information gain difference between *HKMeans* and *TPKMeans* in Figures 7, 8 and 9 clearly indicates the effect of initial cluster centers on the history-based K-means algorithm (see Section 3.3.3 for details). When the history is large and/or dataset (i.e. page size) is small, *HKMeans* may end up choosing outliers as representative tuples. This problem is due to a choice of wrong initial cluster centers. Figure 7 shows that as the page size increases for a fixed sampling rate, the performance of *HKMeans* and *TPKMeans* seem to improve—this is because of reduced history and a large dataset. When the page size is very small, *LKMed* seems to become slightly better as compared to *TPKMeans*—this is due to known problem that K-means are effected by outliers more than K-medoid. By selecting proper initial cluster centers in *TPKMeans*, we are able to make its information gain quite close to the ideal *HKMed* algorithm. From Figure 9 we can see that as the history size increases with the increase in sampling rate, the performance of *TPKMeans* worsens, as compared to both the K-medoids based sampling algorithms. With a high sampling rate, the history size becomes larger, and thus affects the K-means based historical sampling algorithms.

Figure 8 shows that the information gain for all algorithms decreases with the increase in number of dimensions. This decrease is due to curse of dimensionality. As the number of dimensions increases, all the tuples in a page become quite far from each other and thus none of the clustering algorithms are very effective in giving a good overall impression. Even at a high dimension such as six, our sampling algorithms can give an information gain of more than 2.

4.2.3 Effect of Truncated History

Figures 10 and 11 show the effect of truncated history's size on computational performance and information quality. In these experiments, we kept the other parameters at the default values and varied the size of truncated history L.

Figure 10 shows that when we have a very small history, i.e., $L = 1$, the two K-medoids and the three K-means based algorithms have similar computation time. As we increase history, we see that computation time of *HKMed* becomes better than *LKMed*, but as we increase L further, the time to compute the nearest neighbor increases and the computation time of both the algorithm seems to converge. For *HKMed*, the dip in computation cost is due to a reduced amortized cost of the *SwapCost* algorithm, as discussed earlier. The computation time of history based K-means al-

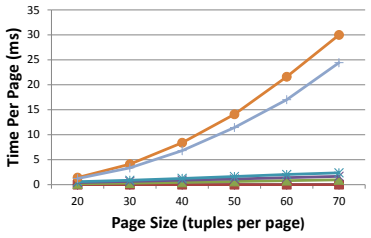

Figure 4: Effect of Page Size

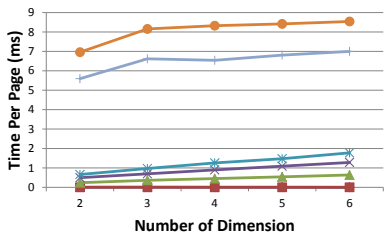

Figure 5: Effect of Dimension

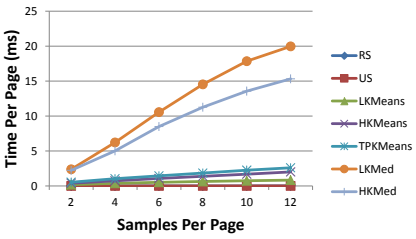

Figure 6: Effect of Sampling Rate

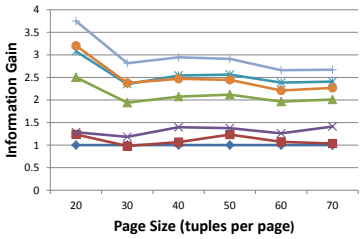

Figure 7: Effect of Page Size

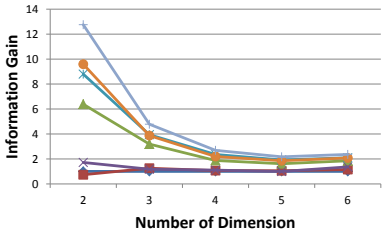

Figure 8: Effect of Dimension

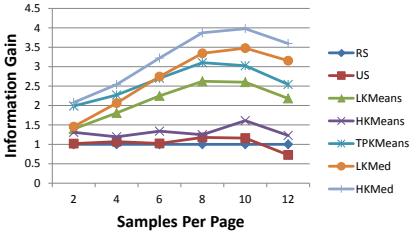

Figure 9: Effect of Sampling Rate

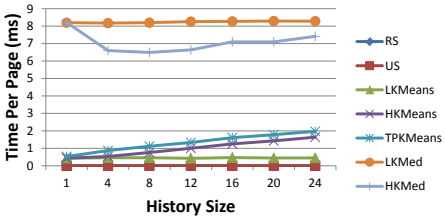

Figure 10: Computation Time

gorithm goes on increasing because of the nearest neighbor computation.

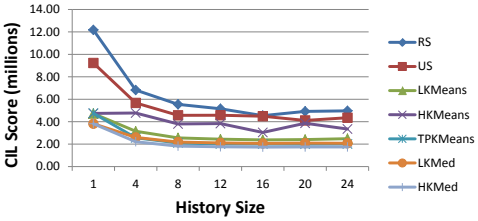

Figure 11: Information Quality

Figure 11, shows that the *CIL* score of all sampling algorithms become stable for L greater than 10 (the information gain also becomes constant because it is ratio of two *CIL* scores). Based on this, we used $L = 10$ for all other experiments. Note that we plotted the *CIL* score in Figure 11 rather than *Information Gain*. We did so to make clear that the flattening of benefits with larger L is not just a flattening of relative benefits between the algorithms, but rather an absolute flattening.

4.3 Summary Recommendations

K-medoids based sampling algorithms are fast enough to support normal browsing in all parameter ranges. Studies have shown that for a user interface to seem instantaneous from a cognitive perspective, the system has a bound of reacting within 100 ms [4]. Our own studies showed that users often spend less than 100ms on a page when they are browsing quickly. Some users spend as little as a few tens of milliseconds on each page. We see from Figures 4, 5 and 6 that the computation time of all proposed sampling algorithms is well within the 100 ms limit. The *LKMed* and *HKMed* algorithms do start to take time that may become noticeable when the page size is large and the sampling rate is very high. In general, most display screens can show 20-40 tuples per page, and when a user scrolls quickly, our system is expected to show only a few tuples per page (i.e a low sampling rate).

In short, we should choose the sampling algorithm based purely on information quality. In other words, *HKMed* can be used for scrolling in most parameter ranges. If computation time becomes a concern, *TPKMeans* is the preferred algorithm since it sacrifices slightly in information quality compared to *HKMed*, but is much faster to compute.

5. USER STUDY

The goal of our user study is to measure the usability of a full display vs. the sampling-based variable-speed scrolling interface. Usability can be measured in terms of users' ease-of-use, efficiency and quality of response to a given task.

We requested 8 users from our university to participate in sessions comprising two similar tasks on each of the three datasets—Stock, Abalone and ImageSegmentation—once using the condensed display and once using the full display. To reduce the effect of learning, we created the following four sequences of sessions that distribute and reorder the tasks, datasets and user interfaces evenly. We asked two users to perform each of the following session sequences:

- **Set0:** S1F, A2S, I1F, S2S, A1F, I2S
- **Set1:** S2S, A1F, I2S, S1F, A2S, I1F
- **Set2:** S2F, A1S, I2F, S1S, A2F, I1S
- **Set3:** S1S, A2F, I1S, S2F, A1S, I2F

For a session code 'XYZ', X indicates the dataset's name i.e., S for STOCK, A for ABALONE, I for IMAGESEGMENTATION; Y indicates task code, i.e, Task 1 or 2 from the dataset X (two tasks were designed for each data set, as we explain below); and Z indicates the display mode, i.e., F for full display and S for condensed display. We indicate the users as $U_0, U_1, ..., U_7$. We gave user U_i the s session sequence $i\%4$. For each session we measured the time taken by the user to finish the task, the quality of user's response and user's scrolling pattern.

For the condensed display we used the *HKMed* sampling algorithm. To measure the quality of our samples, we turned off the full display mode while the users scrolled backward. In our user studies, when the users scrolled backward, we showed those tuples from a page that were shown in the most recent forward scroll through that page. We performed all user studies with page size = 30 tuples/page.

5.1 Interesting Patterns (STOCK)

There are many applications where users are interested in finding trends or patterns, such as increasing or decreasing trends, periods with high and low data variance, cyclical periods, etc., over time series data.

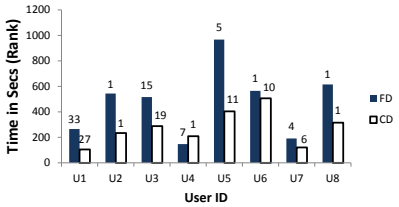

Figure 12: Max-Min Page Variance

We measured the usability of our system in identifying interesting patterns through the STOCK dataset. We asked the users to scroll through the dataset, which was ordered with respect to time, and identify pages with minimum and maximum variance, defined as the difference between the maximum value of `Max Price` column and minimum value of `Min Price` column. Figure 12, shows the time taken by all 8 users using full display (FD) and condensed display (CD) interface. To evaluate the user's response quality, we computed the actual variance of all the pages and then measured the quality of user's response with respect to the actual variance. For example, if user's max variance page is 5th highest in terms of actual max variance, then we say that quality is 5. At the top of each time bar, we have displayed the quality of user's response. The results show that even though the user's response quality is almost similar for both interfaces, users are able to do tasks using our condensed display interface twice as fast as compared to the full display interface.

5.2 Simple Regression Task (ABALONE)

There are many applications where given certain feature measurements, one is interested in finding the expected value for a missing feature. For example, given descriptions of a car or a house on sale, one would like to have an expected guess of what should be a reasonable price.

We measured the usability of our system in assisting regression tasks using the ABALONE dataset. As compared to general housing or automobile datasets, the ABALONE dataset is less noisy and thus it is comparatively easy for

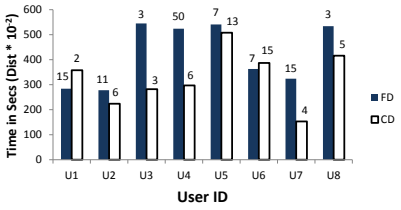

Figure 13: Predicting Missing Features

a user to manually perform the regression task. In this task, we picked two instances as test data from the ABALONE dataset and removed their `length` and `shell weight` features. We sorted the data according to `length`. We asked each user to fill in these two missing features for one instance using the full display interface and other using the condensed display interface. For the user studies, we removed the `diameter` feature from the dataset as this feature was very closely correlated with `length`, and thus users could predict the `length` closely using this feature itself. Other features were not that closely related with the `length` feature. We measured each user's response quality by computing the Manhattan distance of user's response for the two missing features with respect to the actual values. The results are shown in Figure 13. The bars show the time taken by each user and the values on top of the bar indicates the Manhattan distance of user's response to the actual missing values. For most users, both response quality and time are better using the condensed display interface than the full display interface.

Since this task involved a larger number of features and comparatively larger dataset, this was very indicative of how users struggle with large relational query results. In this task, the tuples that were similar to task 1 appeared in the first half of the ordered data, whereas for task 2, they appeared in the second half. In the condensed display interface, users were quick to appreciate a good overall impression of each page through a few selected samples and compare with the test data. We can see that when task 2 was given to users U_3,U_4, U_7 and U_8 using the full display, they took much more time as compared to users U_1,U_2, U_5 and U_6 who had done the same task using condensed display.

5.3 Discriminating Columns (IMAGESEGMN.)

In many applications, a user does not have a priori knowledge of the underlying data. Thus, they try to figure out columns which can help them make discriminating judgements by browsing through the data, e.g. identifying features that can help them the most in classifying houses, cars, hotels etc., at different price ranges. Although each house, car etc., can have a large number of features, one needs to identify features that are important for making decisions, a task that is often best done by visual inspection. By knowing good discriminating features and suitable ranges, a user can either form a new, more precise query or easily predict the class of a new data record.

We evaluated user performance on a task of identifying good discriminating columns using the IMAGESEGMENTATION dataset. This data set has 19 columns, many of which are rather technical in nature. 19 columns is too many for use in a simple scroll without horizontal panning. Therefore, we selected 7 out of 19 attributes that were easy for non-expert users to understand. Specifically, we selected attributes 10–16, i.e., `Mean Intensity`, `Raw Red`, `Raw Blue`,

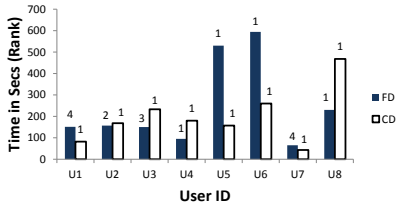

Figure 14: Max Discriminating Column

`Raw Green`, `Excess Red`, `Excess Blue` and `Excess Green`. The were 330 images in the dataset for each of seven different types, such as brickface, sky, and grass. We asked users to find the most discriminating columns for image types: brickface and sky; and brickface and grass. We measured the quality of a discriminating column using the Fischer Linear discriminant.

$$\mathcal{J}(C_k, i, j) = \frac{|m_{ik} - m_{jk}|^2}{{s_{ik}}^2 + {s_{jk}}^2} \quad (8)$$

Here, $\mathcal{J}(C_k, i, j)$ is a measure of column C_k's discriminating ability for class i and class j; m_{ik} and s_{ik} represents the mean and standard deviation of class i in column C_k.

Figure 14 shows the user's response time and quality of answer at top of the time bar. The quality is measured in terms of the true rank of the user's response column using the discriminating score defined in Equation 8. All users were able to identify the best column using condensed display interface, but they made errors using the full display because of noisy data and too much information. Furthermore, in terms of the user's time-to-task, the condensed display interface is superior to the full display interface.

5.4 Relationship between Sampling Rate and Scrolling Speed

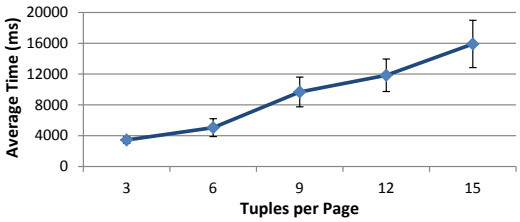

Figure 15: Scrolling Speed Vs. Sampling Rate

The results in this section shows how verbosity affects users reading time. We conducted this study over 6 users. Each user was asked to scroll through 5 pages of a relation, where each page had 30 tuples and 5 columns, and each cell of the relation was randomly assigned an integer value between 0 and 999. Users were tasked to locate a predetermined value on each page of the dataset (this forced the user to read each page completely). In Figure 15, we plot the average time spent per page and standard error, while varying the verbosity of our display from 3 to 15 tuples per page. In addition to the linear, inverse relationship between reading speed and tuples-per-page, we also observe that users take approximately 1 second to read a tuple on average for our dataset. When browsing quickly, users are often not looking at all the data values—rather, they may focus on the single sorted attribute. In such cases, they can browse much faster, possibly spending less than 100 ms per tuple. From Figure 15, one can see that as verbosity increases, the user takes more time to read a page.

5.5 Users' Feedback

Since it is hard to objectively measure the ease-of-use of a system, we polled our test users for comments. All said that it was much less stressful to use the condensed display as compared to full display interface.

6. RELATED WORK

The need for fast browsing has been established through numerous user interface studies [14, 6, 8, 7], done in the area of Human Computer Interaction. A primary difference between our work and a majority of previous work is that existing systems do not take into consideration the user-specific (attribute-wise) sort preference, which is a very typical browsing pattern in databases. Moreover, we will see that the the related algorithms, discussed below, statically determine their full result set. Due to their static nature, even if a user varies her browsing speed at a certain page, we cannot vary the results. Our interface provides best possible non-redundant, overall information of any page based on user's current browsing behavior. We show those tuples that give the best overview, and not necessarily ranked tuples.

Top-K Result Set: Evaluating top-K query has been an important research aspect in all areas of information retrieval. Many results may satisfy a user's query. But by ranking the result set, these ranking algorithms enable a user to quickly locate the desired information by just browsing the top few results. For relational data, top-K ranking algorithms have been proposed in [2, 9].

Top-K Diverse Set: It is hard to design an automatic top-K ranking algorithm that can simultaneously satisfy the information need of all users, because rankings change with change in user preference. To satisfy most users, algorithms for computing top-K diverse set has been proposed, for example web queries [1] and SQL queries [10].

Data Summarization: Large data sets are often browsed through data summarization techniques, such as clustering and faceting. Different types of data mining algorithms, including clustering, is very nicely presented in [15]. Faceted search, or guided navigation, is very useful in enhancing user's browsing experience, in which users can browse the data along various "facets" or attributes. Faceted search interface [12, 11] were traditionally used for text and/or image data. For relational data, Senjuti et al. [5] have proposed algorithms for identifying important facets and the sequence in which these facets should be presented to a user. Identifying important facets enable a user to quickly locate the desired information.

Sampling: Sampling algorithms have been extensively used in databases for various reasons, such as identifying important statistical information [26], fast query processing [22], or computing approximate query result [3]. In the context of browsing, the precise problem formulation we require is slightly different.

Information Visualization: There are many visualization tools, such as tag clouds [20, 19], which enable a user to quickly locate important information. These tools signal important information through distinct text properties, such as font, color etc.

User Interfaces: For relational data there are fast browsing interfaces, such as, MusiqLens [23] and DataScope [27]. These interfaces cluster the whole query result and then

present the result in a predetermined manner. In these, the order in which information is presented does not change with user's dynamic browsing pattern. In our system, we take into account the information that user has already seen and the current scrolling rate, to determine how much and which new information to show.

Our system has similar motivations as that of variable-speed scrolling system, proposed by Igarashi et al. [16], for browsing large text documents. While browsing text documents, our eyes follow visual hints, such as different font sizes, section headings, graphics, knowledge of structural outline etc. When a user scrolls too fast through a text document even the prominent visual hints start getting distorted and the user loses track of her position in the document. In [16], the authors assume that there is a maximum threshold scrolling speed beyond which things would start getting distorted. When a user scrolls below the threshold speed, their system shows full information, but when the scroll speed is beyond the threshold speed, the system starts showing only important markers and hides the finer details. The markers are easy to identify in text documents because of different font sizes, section headings etc. They had also tried to use their technique for structured data, such as dictionaries, but they found out that it was not useful because everything was very homogenous and markers were very hard to find.

7. CONCLUSIONS

In this paper, we demonstrated how the user's browsing of structured query results can be supported by providing an interface mechanism to rapidly scroll through query results. We implemented a scrolling interface that varies the verbosity of information based on the speed of scrolling. We discussed several scrolling-aware algorithms that can select representative tuples of a higher quality as compared to random sampling. Our approach reduces the information load of the user and provides a quick overview of the data through few representatives. We formally quantified the metric of information loss while browsing structured data, and demonstrated, through extensive user study and experimental evaluation, that our variable-speed scrolling interface provides a better browsing experience.

Scrolling interfaces are very commonly used in hand-held devices, such as cell phones etc., which have very small display unit and thus most users would select only 2-3 dimensions. For such small form-factor displays, our sampling algorithms, such as *HKMed*, are even more beneficial, providing 10–15 times information gain as compared to random sampling.

Acknowledgement

This work is supported in part by NSF under grant IIS-1017296. We would also like to thank Bin Liu and Honglak Lee for their insightful comments on this work.

8. REFERENCES

[1] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. *WSDM*, 2009.

[2] S. Agrawal and S. Chaudhuri. Automated ranking of database query results. *CIDR*, 2003.

[3] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. *SIGMOD*, 2003.

[4] B. Bailey et al. The Effects of Interruptions on Task Performance in the User Interface. *INTERACT*, 2001.

[5] S. Basu Roy, H. Wang, G. Das, U. Nambiar, and M. Mohania. Minimum-effort driven dynamic faceted search in structured databases. *CIKM*, 2008.

[6] M. Bates. Subject access in online catalogs: A design model. *ASIS J.*, 1986.

[7] M. Bates. The design of browsing and berrypicking techniques for the online search interface. *Online Information Review*, 1989.

[8] N. Belkin, R. Oddy, and H. Brooks. Ask for information retrieval. *Journal of Documentation*, 1982.

[9] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. *VLDB*, 2004.

[10] Z. Chen and T. Li. Addressing diverse user preferences in SQL-query-result navigation. *SIGMOD*, 2007.

[11] W. Dakka, P. Ipeirotis, and K. Wood. Automatic construction of multifaceted browsing interfaces. *CIKM*, 2005.

[12] J. English, M. Hearst, R. Sinha, K. Swearingen, and K. Yee. Hierarchical faceted metadata in site search interfaces. *CHI*, 2002.

[13] A. Frank and A. Asuncion. UCI Machine Learning Repository, 2010.

[14] A. Goodchild. An evaluation scheme for trader user interfaces. *IFIP*, 1995.

[15] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

[16] T. Igarashi and K. Hinckley. Speed-dependent automatic zooming for browsing large documents. *UIST*, 2000.

[17] L. Kaufman, P. Rousseeuw, and E. Corporation. *Finding groups in data: an introduction to cluster analysis*. John Wiley, 1990.

[18] E. Keogh, X. Xi, L. Wei, and C. A. Ratanamahatana. The UCR Time Series Homepage, 2006.

[19] G. Koutrika, Z. Zadeh, and H. Garcia-Molina. Data clouds: summarizing keyword search results over structured data. *EDBT*, 2009.

[20] B. Kuo, T. Hentrich, B. Good, et al. Tag clouds for summarizing web search results. *WWW*, 2007.

[21] C. Li, M. Wang, L. Lim, H. Wang, and K. Chang. Supporting ranking and clustering as generalized order-by and group-by. *SIGMOD*, 2007.

[22] R. Lipton, J. Naughton, D. Schneider, and S. Seshadri. Efficient sampling strategies for relational database operations. *Theoretical Computer Science*, 1993.

[23] B. Liu and H. Jagadish. Using trees to depict a forest. *VLDB*, 2009.

[24] J. MacQueen et al. Some methods for classification of multivariate observations. *BSMSP*, 1967.

[25] R. Ng and J. Han. A method for clustering objects for spatial data mining. *TKDE*, 2002.

[26] F. Olken and D. Rotem. Simple random sampling from relational databases. *VLDB*, 1986.

[27] T. Wu, X. Li, D. Xin, J. Han, J. Lee, and R. Redder. DataScope: viewing database contents in Google Maps' way. *VLDB*, 2007.