Using Trees to Depict a Forest

Bin Liu
Department of EECS
University of Michigan
Ann Arbor, USA
binliu@umich.edu

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

ABSTRACT

When a database query has a large number of results, the user can only be shown one page of results at a time. One popular approach is to rank results such that the "best" results appear first. However, standard database query results comprise a set of tuples, with no associated ranking. It is typical to allow users the ability to sort results on selected attributes, but no actual ranking is defined.

An alternative approach to the first page is not to try to show the best results, but instead to help users learn what is available in the whole result set and direct them to finding what they need. In this paper, we demonstrate through a user study that a page comprising one representative from each of k clusters (generated through a k-medoid clustering) is superior to multiple alternative candidate methods for generating representatives of a data set.

Users often refine query specifications based on returned results. Traditional clustering may lead to completely new representatives after a refinement step. Furthermore, clustering can be computationally expensive. We propose a tree-based method for efficiently generating the representatives, and smoothly adapting them with query refinement. Experiments show that our algorithms outperform the state-of-the-art in both result quality and efficiency.

1. INTRODUCTION

1.1 Motivation

Database queries often return hundreds, even thousands, of tuples in the query result. In interactive use, only a small fraction of these will fit on one display screen. This paper studies the problem of how best to present these results to the user.

The "Many-Answers Problem" has been well documented [5]: too many results are returned for a query that is not very selective. This problem arises because: i) it is very difficult for a user, without knowing the data, to specify a query that returns *enough* but not *excessive* results; and ii) often a user starts exploring a dataset without an exact goal, which becomes increasingly clear as she learns what is available. Consider Example 1 below, where a user searches a used car database for a Honda Civic.

Example 1. Ann wants to buy a car, and visits a web site for used cars. The web site is backed by a database that we simplify for this example to have only one table "Cars" with attributes *ID*, *Model*, *Price*, and *Mileage*. Ann specifies her requirements through a form on the web site, resulting in the following query to the database: `Select * from Cars where Model = 'Civic' and Price` $<$ `15,000 and Mileage` $<$ `80,000.` The query she formulates may have thousands of results since it is on a popular model with unselective conditions. How should the web site show these results to Ann?

A common approach to displaying many results is to batch them into "pages". The user is shown the first page, and can navigate to additional pages as desired, and "browse" through the result set. For this process to make sense, the results must be organized in some manner that the user understands. One popular solution is to sort the results, say by Price or Mileage in our example. However, this sorting can be computationally expensive for large result sets. More important, similar results can be distributed many pages apart. For example, a car costing 8500 with 49000 miles may be very similar to another costing 8200 with 55000 miles, but there could be many intervening cars in the sort order, say by price, that are very different in other attributes (e.g. high mileage but recent model year, high mileage but more features, low mileage but in an accident, and so on).

Another possibility is to order results by what the system believes is likely to be of greatest interest to the user. Indeed, there is a stream of work [9] trying to develop ranking mechanisms such that the "best" results appear first. Such techniques can be successful when the system has a reasonable estimate of the user's preference function. However, determining this can be hard: in our example the system has no way to tell what Ann's tradeoff is for price versus mileage, let alone other attributes not even mentioned.

This "Many-Answer Problem" has also attracted much attention from the information retrieval community. The importance of the first page of results for a search interface has been well documented [2, 13]. It has been shown that over 85% of the users look at only the first page of results returned by a search engine. If there is no exact answer in the first page to meet users' information need, the first page needs to deliver a strong message that there are interesting results in the remaining pages.

VLDB '09, August 24-28, 2009, Lyon, France

ID	Model	Price	Mileage	*Zoom-in*
643	Civic	14,500	35,000	311 more Cars like this
876	Civic	13,500	42,000	217 more Cars like this
321	Civic	12,100	53,000	156 more Cars like this
452	Civic	11,200	63,000	87 more Cars like this
765	Civic	10,200	71,000	65 more Cars like this
235	Civic	9,000	78,000	43 more Cars like this

Figure 1: MusiqLens Example

ID	Model	Price	Mileage	*Zoom-in*
643	Civic	14,500	35,000	71 more Cars like this
943	Civic	14,900	25,000	63 more Cars like this
987	Civic	14,700	28,000	55 more Cars like this
121	Civic	14,300	40,000	45 more Cars like this
993	Civic	14,100	43,000	40 more Cars like this
937	Civic	13,900	47,000	37 more Cars like this

Figure 2: After Zooming on First Tuple

In this paper, we solve the "Many-Answer Problem" starting from a user's point of view. Psychological studies have long shown that human beings are very capable of learning from examples and generalizing from the examples to similar objects [24, 30, 28]. In a database querying context, the first screen of data can be treated as examples of a large dataset. Since users can expect more items similar to the examples, we should make them as representative as possible.

To accomplish the above task, we propose a framework called *MusiqLens*, as part of the MUSIQ project [1] from the University of Michigan. MusiqLens is designed to: i) automatically displays the best *representatives* result tuples in the first screen of results when the result set is large, ii) at user's request, displays more representatives similar to a particular tuple, and iii) automatically adapt to user's subsequent query operations (selections and joins). This is exemplified in Fig. 1. Notice that each tuple represents many cars with similar Price and Mileage. The representatives naturally fragment the whole dataset into clusters such that cars of various price and mileage range are shown. The representatives themselves have a high probability of being what the users want. If they are not, they can lead to more similar items. On the right side of each representative tuple, the number of similar items is displayed. A hyper-link is provided for the user to browse those items. Suppose now the user chooses to see more cars like the first one. Since they cannot fit in one screen, MusiqLens shows representatives from the subset of cars (Fig. 2). We call this operation "zooming-in", in analogy to zooming into finer level of details when viewing an image. After seeing the first screen of results, if the user now has more confidence to further lower the price condition (since there are more than 100 cars with price around $10k), she could add a condition `price < 10,000`. The next screen of results are generated with the same spirit. By always showing the best representatives from the data, we enable users to quickly learn what is available in the data without actually seeing all the tuples. We have built a prototype of MusiqLens. See [18] for a demonstration. [1]

1.2 Challenges

Several challenges must be addressed before one can construct an effective interface such as the one shown in Fig. 1. We discuss these below. Let the first page of results be limited to k tuples. We call these tuples on the first page *representatives* of the whole result set.

Representation Modeling Our first problem is to determine what it means for a small set of points to "represent" a much larger data set. How can we choose between two (or more) choices of possible representatives? Although it is generally accepted that humans can learn from examples, to our knowledge there is no gold standard for generating those examples.

A naive approach is to display results sorted by some attributes. This approach only presents to users a very small fraction of results at the boundary of the value domain and makes it impossible to find other tuples (for example, a car that balances the price and mileage). Should we uniformly sample k tuples from the results? While this can reflect the density of data distribution, it misses small clusters that may interest the user. Should we sample the results using density biased sampling [25] instead? We need to answer these questions and find a metric that matches human information seeking behavior.

Representative Finding Challenge Once the representation model is decided, we need to efficiently find representatives for the result set that are "best" in this model. MusiqLens will impose some overhead, but the waiting time perceived by the user should not be significant relative to the time the database server needs to finish the query.

Query-Refinement Challenge In the application scenarios of interest to us, such as a used car purchase or a hotel booking, users are typically exploring available options. Queries will frequently be modified and reissued, based on results seen so far. For example, Ann may decide to restrict her search to cars with less than 60,000 miles (instead of the 80,000 originally specified). In addition, once we show representative data points we should permit users to ask for "more like this," an operation we call *zooming in*. See Fig. 2. Such operations must be fast, which means that we can probably not afford to recompute representatives from scratch.

1.3 Contributions

Our first contribution is the MusiqLens framework for solving the "Many-Answers Problem" in database search. We propose to generate best representatives from a result set to show on the first result page. Based on the representatives, users can obtain a global and diversified view of what is available in the data set. Users can drill down by choosing to view more items similar to any tuple in the screen.

Our second contribution is the development of a representation model and metric. Since the ultimate purpose is for users to learn about the data, we compared several popular candidates with a user study. Results are reported in Sec. 2, and show that k-medoid clustering with minimum average distance to be the technique of choice.

The third contribution is a fast algorithm to find repre-

[1]Note that *MusiqLens* was named *DataLens* in the demonstration paper.

sentative data points. Based on the cover-tree structure, we are able to generate high-quality k-medoids clusters, for metrics of average-distance or max-distance. This algorithm is presented in Sec. 3. Experiments show the distance cost and computational cost are both superior over the state-of-the-art.

The fourth major contribution is algorithms for maintaining representative samples under common query operations. When a query is applied, some of the original samples may still qualify to be in the answers and some are not. How to generate new representative samples without rebuilding the index from scratch? We devised algorithms for handling selection and projection operators such that we always have a valid cover tree index, and we can incrementally adjust the set of representatives in response to query refinement. These algorithms are presented in Sec. 4.

Our final contribution is a thorough experimental study of our algorithms, compared with the state-of-the-art competitor (R-tree based algorithm), presented in Sec. 5. Experiments show that: i) for generating initial representatives, we achieve better quality results (in terms of distance metric) in shorter time, and ii) our algorithms can adapt to selection and projection queries efficiently while R-tree based algorithms cannot.

2. WHAT IS A GOOD SET OF REPRESENTATIVES

Given a large data set, our problem is to find a small number of tuples that best represent the whole data set. In this section, we evaluate various options. Note that statistical measures, such as mean, variance, skew, moments, and a myriad of more sophisticated measures, can be used to characterize data sets. While such measures can be important in some situations, we believe they are not suitable for lay users interested in data exploration. Even for technically sophisticated people like members of our community, a few sample hotel choices convey a much better subjective impression than statistical measures of price and location distributions. As such, we only consider using selected tuples from the actual result set as a representative set.

2.1 Candidate Representative Choices

We consider the following methods for choosing representatives:

1. **Random selection**. Generate uniformly distributed random numbers in the range of [1, < Data Set Cardinality >] and use them as index to select cars as samples. This is a baseline against which to compare other techniques.

2. **Density biased sampling**. It is argued that uniform sampling favors large clusters in the data and may miss small clusters. We therefore use the algorithms by Palmer and Faloutsos [25] to probabilistically undersample dense regions and over-sample sparse regions.

3. **Select k-medoids**. A *medoid* of a cluster of data points is the one whose average or maximum dissimilarity is the smallest to other points. We denote the two kinds of medoids as *avg-medoid* and *max-medoid*, respectively. Under the most commonly used Euclidean distance metric, we select k avg-medoids and max-medoids from the data. Note that k-means clustering is frequently used, and is very similar. We do not consider that since the mean values obtained may not represent actual data points, and so may mislead users.

4. **Sort by attributes**. Since sorting is the standard facility provided in systems today, we consider this choice as well. We note that sorting is one attribute at a time in a multi-attribute scenario.

5. **Sort by typicality**. Hua et al. [12] proposed to generate the most "typical" examples. They view all data as independent identically distributed samples of a continuous random variable, and they select data points where the probability density function (estimated from the data) has highest values.

In the rest of the paper we use the following abbreviations for each method: Random (random samples) , Density (density-biased sampling), Avg-Med (avg-medoids), Max-Med (max-medoids), Sort-<attr> (sorted by attribute <attr>), and Typical (sorted by typicality).

2.2 Data

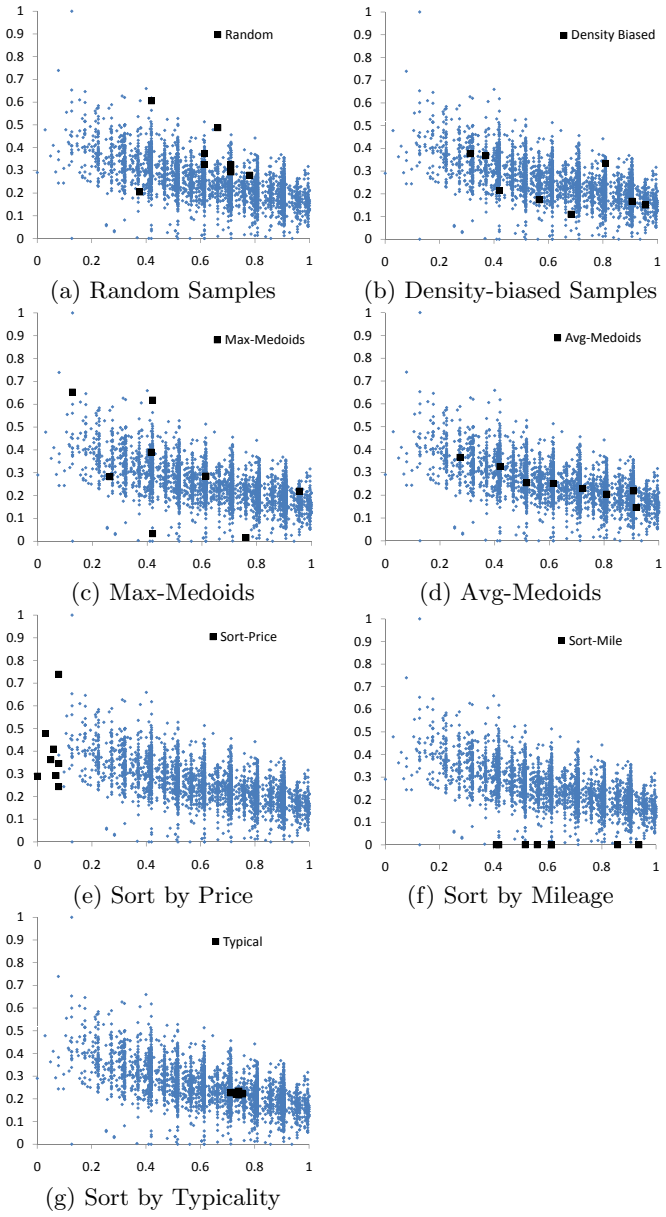
We obtained information about cars of model Honda Civic from Yahoo! Autos. For each car, we have values for numerical attributes Mileage and Price. The site limits the total number of results for a particular type of car to 4100 items, some of which do not have mileage or price information and are removed. This leaves us with 3922 cars that we used in our study.

In Fig. 3 we show representatives generated using all above methods (note that all data have been normalized to range [0, 1]). The whole dataset is shown in the background of each figure. We can see visually that sorting does poorly, whether we sort first by price or by mileage. Even sorting by typicality does poorly, giving us a few points near the "center", but no sense of where else there may be data points. We also see that Avg-Medoids, Max-Medoids and Density-biased Samples all appear to do much better than random samples. We further see that Max-Medoids seems to choose points that delineate the boundary of the data set whereas Avg-Medoids gives us points in the "center" of it, with density-biased samples somewhere in between.

2.3 User Study

The goal of choosing representative points is to give users a good sense of what else to expect in the data. While each of us can form a subjective impression of which scheme is better by looking at Fig. 3, we would like to verify this through a careful user study. Towards this end, we recruited 10 subjects, and showed them the seven sets of representative points in random order, without showing them anything else about the data, and not telling them that these were all for the same distribution. For each set of representatives, we sought to elicit from the users what the rest of the data set may look like.

Eliciting information regarding an imagined distribution is very tricky. We cannot get into the head of the user! After considering many alternatives, we settled on asking the users to suggest a few additional points that they would expect to see in the data set. (We required these points not to be "too close" to the representatives provided or to one another – but all of our subjects naturally adopted this constraint without explicit direction from us). We require that



(a) Random Samples (b) Density-biased Samples

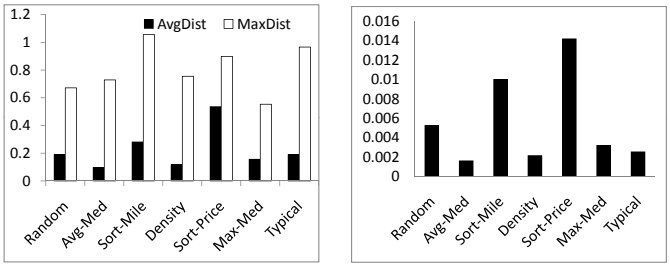
(c) Max-Medoids (d) Avg-Medoids

(e) Sort by Price (f) Sort by Mileage

(g) Sort by Typicality

Figure 3: Samples Generated Using Different Methods. Light points are actual data, and dark points are generated samples.

the points suggested by the user can not be any existing point. Given a set of predicted data points, we can measure how far these predictions are from actual data set values. For each point in the dataset, we find the distance to the closest point in the predicted set. We call this the *prediction error distance* for that data point. If the minimum prediction error distance is small, that tells us that an individual predicted point is good, but says nothing about the overall data set. If the maximum prediction error distance is small, that tells us that there is no very poor prediction – the user has not been misled about the shape of the data set. Finally, if the average prediction error distance is small, that gives us a global metric of how well the set of predicted points as a whole match the actual data set. We refer to these three metrics as MinDist, MaxDist, and AvgDist, respectively. We computed values for all three, averaged across all participants. The results of AvgDist and MaxDist are shown in Fig. 4 (a), and MinDist is shown in Fig. 4 (b) using a different scale in the y-axis.



(a) AvgDist and MaxDist (b) MinDist

Figure 4: Average Distance Results for the Seven Methods

In Fig. 4 (a), avg-medoids (Avg-Med) stands out as the best based on AvgDist measurement, while max-medoids (Max-Med) is the best in MaxDist measurement. For MinDist measurement (Fig. 4 (b)), the winner is not clear. Among the two best choices, avg-medoids has a smaller value than density-biased sampling (0.00161 vs. 0.00253). However, the values are too small to be statistically significant. We calculated the statistical significance using Mann-Whitney test to verify the above observation. p-values are shown in Table 1. The first row shows the p-values of Avg-Med against others under the AvgDist metric, second row shows that of Max-Med against others under MaxDist metric, and the third row shows Avg-Med against others under MinDist metric. All values are significant, except one – Avg-Med vs. Density under MinDist metric, meaning that the two are similar in performance. Since Avg-Med is clearly better than Density under AvgDist metric, it is overall more desirable. In summary, if we consider AvgDist and MinDist metric, avg-medoids is the choice; if we consider MaxDist, max-medoids is the best.

The conclusion from the investigation described above is that k-medoid (average) cluster centers constitute the representative set of choice. k-medoid (maximum) cluster centers may also make sense in a few scenarios. Even though the rest of the paper will focus only on the former, computation of the latter is not that much different, and the requisite small changes are not hard to work out. For the rest of the paper, we refer to average medoids when we use the term *medoid*. Formally, for a set of objects $\boldsymbol{O}$, *k-medoids* are a subset $\boldsymbol{M}$ from $\boldsymbol{O}$ with k objects, which minimize the average distance from each point in $\boldsymbol{O}$ to the closest point in $\boldsymbol{M}$.

3. COVER-TREE BASED CLUSTERING ALGORITHM

Clustering has been studied extensively. Many clever techniques have been developed, both to cluster data sets from scratch and to cluster with the benefit of an index. See Sec. 6 for a short survey. Unfortunately, none of these techniques address the query-refinement challenge or even support in-

Table 1: p-value of Mann-Whitney Test

	Random	Avg-Med	Sort-Mile	Density	Sort-Price	Max-Med	Typical
AvgDist, Avg-Med vs.	<0.0001	NA	<0.0001	0.0087	<0.0001	<0.0001	<0.0001
MaxDist, Max-Med vs.	0.0228	<0.0001	<0.0001	<0.0001	<0.0001	NA	<0.0001
MinDist, Avg-Med vs.	0.0011	NA	0.0018	**0.1922**	<0.0001	0.0104	0.0446

cremental recomputation. As such, we must develop a new algorithm to meet our needs.

We propose using the cover-tree [4] data structure for clustering. The properties of cover-tree (which will be discussed shortly) make it a great structure for sampling. This immediately reduces the problem of finding medoids from the original data set to finding medoids in the sample. We then use statistics gathered during the tree construction phase to help find a good set of medoids. We begin by providing some brief background on the cover-tree in Sec. 3.1, followed by our novel contributions in the subsequent sub-sections.

3.1 Cover-tree

Cover-tree was proposed by Beygelzimer, Kakade, and Langford in 2006 [4]. It is so named because each level of the tree is a "cover" for the level beneath it. For convenience in explanation, we assume that the distance between any two data points is less than 1 (we will see later how this condition can be relaxed). Following convention, we number the levels of the tree from 0 (root level). For level i, we denote the value of $1/2^i$ as *D(i)*, which is a monotonically decreasing function of i. The condition that distance between any two points is less than 1 can be relaxed if we allow i to be negative integers. A cover-tree on a data set S has the following properties for all levels $i \geq 0$:

1. Each node of the tree is associated with one of the data points s_j.
2. If a node is associated with data point s_j, then one of its children must also be associated with s_j (nesting).
3. All nodes at level i are at separated by at least $D(i)$ (separation).
4. Each node at level i is within distance $D(i)$ to its children in level $i+1$ (covering).

Fig. 5 shows a cover-tree of scale 2 for data points s_1 to s_7 in 2-dimensional space. Nesting property is satisfied by repeating every node in each lower level after it first appears. Covering property ensures that nodes are close enough to their children. Separation property means that nodes at higher levels are more separated (e.g., nodes s_1, s_4, s_5 are far away from root node s_7). Points s_1 and s_2 are at a larger distance from each other than are s_5 and s_6. Thus s_2 is a child of $s1$ at level 2 while $s6$ is a child of s_5 at level 3 (where inter-node distance is smaller). We can prove that the distance from any descendant to a node at level i is at most $2 \times D(i)$ (using the convergence property of $D(i)$). Cover-tree naturally provides a hierarchical view of the data based on the distance among nodes, which our algorithms will exploit.

The cover-tree shown in Fig. 5 is a theoretical *implicit* tree, where every node is shown. It is neither efficient nor necessary to repeat a node when it is the lone child of itself in intermediate levels (for example, s_7 at level 1 and 2). In practice, we use an *explicit* tree, where such nodes are pruned. So every explicit node either has a parent other than itself or or a child other than a self-child. We call the rest of the nodes *naive nodes*. We use the implicit cover-tree throughout this paper for the ease of understanding. All algorithms in this paper can be easily adapted to the explicit tree.

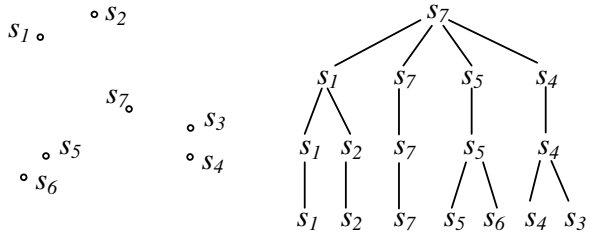


Figure 5: Cover Tree Example

A very important property of cover-tree is that the subtree under a node spans to a distance of at most $2 \times D(i)$, where i is the level at which the node appears. We call this distance the *span* of the node. For example, point s_5 first appears in level 1. The actual span of s_5, however, is best obtained when it appears again at level 2, where it has a non-self child. In both levels, we are in fact trying to get the range of the same subtree. The span obtained at level 2 is half of that obtained at level 1, and it gives more accurate information about the subtree. In the rest of the paper, we always use *span* to refer to the least possible span that can be obtained for the subtree, which is achieved by descending from the root of the subtree to the node that has a non-self child.

The explicit cover-tree has a space cost to $O(n)$, and it can be constructed in $O(n^2)$ time. The tree is bounded in width (number of children of any node) and depth. For more details regarding properties and algorithms in cover tree, we refer the readers to the original paper [4] since they are out of the scope of this paper.

3.2 Using the Cover Tree

3.2.1 Additional Statistics

In order to better grasp the distribution of data, we need to gather some additional statistics of the subtree rooted at each node s_i:

- **Density**. This is the total number of data points in the subtree rooted at node s_i. Note that this includes all descendants of the node. For all nodes at the same level in the cover tree, a larger density indicates that the region covered by the node is more densely populated with data points.
- **Centroid**. This is the average value of all data points in the subtree. Assume that there are T points in total in the subtree. For node s_i, if we denote the

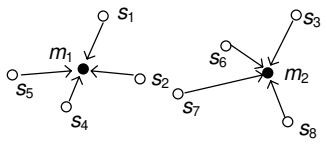


Figure 6: Distance Cost Estimation

N-dimensional points in the subtree as $\overrightarrow{X}_j$ where $j = 1, 2, ..., T$, then $Centroid(i) = \frac{\sum_{j=1}^{T} \overrightarrow{X}_j}{T}$.

We refer to the density and centroid for the subtree under node s as $DS(s)$ and $CT(s)$, respectively. The exact use of density and centroid of a node will become apparent in later sections. Both values can be collected when the tree is being built. As each point is inserted, we increase the density for all its ancestors. Assume the new data point inserted is $\overrightarrow{X}_j$, then for each node i along the insertion path of the new point, we update the density and centroid as follows:

$$DS(s)' = DS(s) + 1$$

$$CT(s)' = \frac{CT(s) \times DS(s) + \overrightarrow{X}_j}{DS(s) + 1}$$

Both operations can be accomplished with a minor change in the recursive insertion algorithm of the cover tree [4].

3.2.2 Distance Cost Estimation of Candidate k-medoid

Using density and centroid information, we can obtain an estimate of the average distance cost for a set of candidate k-medoids, using any level of nodes in the cover tree, without having to read the whole data set. We illustrate using the example in Fig. 6, where we have 8 nodes (s_1 to s_8) and two medoids (m_1 and m_2). Note that each node actually represents many other data points in the subtree. Also, these 8 nodes should form a cover of the tree - they should be *all* the nodes in a certain level of the cover tree. An arrow means the node is closest to the pointed medoid. The total number of data points can be found by summing up the density of each node. Since we do not want to read all data points under a subtree, we use the centroid maintained at the root to stand for the actual data points when calculating the distance to the medoid. For example, to calculate the total distance from all data points under node s_1, we compute the distance from the centroid of s_1 to m_1, and multiply it by its density. We do the same for all other nodes and sum up the total distance. This value is then averaged over the total number of points, and we have obtained an estimate of the average distance cost.

3.3 Average-medoids Computation

We now introduce our algorithm for k average medoid queries. We start by traversing the cover tree from the root until we reach a level with more than k nodes. Assuming this is at level number m, and there are t nodes in this level of the tree. Following notations introduced earlier, we refer to the set of nodes at level m as C_m. For convenience we call this level of the tree the *working level*, since we find medoids by considering primarily nodes in this level. The separation property of the cover tree ensures that nodes in C_m spread out properly. We can view data under each subtree as a small cluster, whose centroid and density are maintained in the root of the subtree. In most cases, m does not equal k. The general approach in the literature is to group C_m into k groups first, and then find the medoid for each cluster. Usually, k *seeds* are first selected among the nodes, and the rest of the nodes are assigned to the respective closest seed. k-medoid clustering is NP-hard [10], so we usually try to achieve a good local minimum in terms of distance cost from data points to their medoids. There are two existing seeding methods:

- **Random**. We can randomly choose k nodes to be the seeds. This is the simplest method with the lowest cost.
- **Space-filling curves**. Hilbert space-filling curve has been shown to preserve the locality of multidimensional objects when they are mapped to linear space [19], a property which Mouratidis et al. [22] exploited in their R-tree based clustering technique. We can apply the same idea in the cover tree. Nodes in C_m could be sorted by Hilbert values, and k seeds chosen evenly in the sorted list of nodes.

Seeds that are not properly chosen may lead the algorithms to a local minimum with high cost. In this paper, we use information provided by the cover tree to choose seeds in a better way than the above techniques. Level $m-1$ of the cover tree, which contains less than k nodes, provides hints for seeds because of the clustering property of the tree. As usual, we denote nodes at level $m-1$ as set C_{m-1}. Intuitively, nodes in C_m that share a common parent in C_{m-1} form a small cluster themselves. When we choose seeds, we should avoid choosing two seeds in one such cluster. Since C_{m-1} contains fewer than k nodes, we will not have enough seeds if we simply choose one node from all that share a parent. As introduced in Sec. 3.1, nodes in C_{m-1} may have different maximum distance to their descendant. As a heuristic, we choose more seeds from children of a node whose descendants span a larger area. Meanwhile, nodes with a relatively small number of descendants should have low priority in becoming a seed, since the possible contribution to the distance cost is small. The contribution of a subtree to the distance cost is proportional to the product of the density and span. We denote this special value as the *weight* of a node. Based on this observation, we use a priority queue to choose seeds as follows. The key of the priority queue is the weight of a node. Initially all non-naive nodes in C_{m-1} are pushed to the queue. We pop the head node from the queue and fetch all its children. We first make sure the queue has k nodes by adding children of the node with largest weight. Afterwards, if any child has a larger weight than the minimum weight of all nodes in the queue, we push it to the queue. We repeat this process until no more children can be pushed into the queue. The first k nodes in the queue are our seeds. This procedure, `CoverTreeSeeding()`, is shown in Algorithm 1.

Once the seeds are chosen, the rest of the nodes are assigned to their respective closest seed to form k initial clusters. We can obtain the centroids of each cluster by computing the geometric center of all nodes from their density and centroid. Using each centroid as input, we can find the corresponding medoid with a nearest neighbor query, which is efficiently supported by the cover tree. For each final medoid o, we call nodes in the working level that are closest

to o as its *CloseSet*. In the future, if the user adds a selection condition and removes a large portion of the CloseSet, the corresponding medoid may have to be eliminated. More details on how the nodes in the CloseSet affects the existence of the medoid are in Sec. 4.2.

Algorithm 1 Cover Tree-based Seeding Algorithm

Input: S: list of nodes in level m of the cover tree
Input: T: list of nodes in level m-1 of the cover tree
Input: k: number of medoids to compute
Input: Q: priority queue for nodes with key being the weight of a node
Output: O: list of seeds

```
minWeight = 0 {maintains the minimum weight of all nodes in Q}
for node t in T do
  if t is a naive node or leaf node then
    T = T − t
  else
    Insert(Q, t)
    if weight(t) < minWeight then
      minWeight = weight(t)
    end if
  end if
end for
repeat
  n = ExtractMax(Q)
  boolean STOP = TRUE
  if Size(Q) < k then
    add all children of n to Q
    update minWeight to smallest weight values seen
    STOP = FALSE
  else
    for each child node c of n do
      if weight(c) > minWeight then
        Insert(Q, c)
        STOP = FALSE
      end if
    end for
  end if
until STOP
O = Exact the first k nodes from Q
```

Optionally, we can try to improve the clusters before computing the final medoids. In the literature [20, 16], usually a repeated updating process is carried out: the centroid of each cluster is updated; nodes are re-assigned to the updated centroids. This process repeats until stable centroids are found. In this process, we can take into account the weight of each node, similar to [16]. This procedure is outline in Algorithm 2. As another refinement step, we can use cover-tree directed swaps. Literature [23] suggests that we can swap a medoid with other nodes and see if this leads to a lower cost. Usually it is the step that computes the cost that is expensive. We have at our disposal a formula that can estimate the distance cost, as described in Sec. 3.2.2. Instead of swapping with a random node, we can swap with nodes that was assigned to the closest neighbor medoid. Both measures significantly cut the computational cost. The details are omitted here due to limited available space.

4. QUERY REFINEMENT

Algorithm 2 Compute Medoids

Input: S: list of nodes in level m of the cover tree
Input: L: list of seeds
Input: k: number of medoids to compute
Output: M: list of medoids

```
for node s in list S do
  assign s to the seed in L whose centroid is closest {forming the initial clusters}
end for{denote the initial clusters as C}
repeat
  for c_i in C, i ∈ [1, k] do
    m_i = the node in c_i closest to geometric center of c_i
  end for
  for node s in list S do
    assign s to closest medoid in M
  end for
until no change in M
```

In practical dataset browsing and searching scenarios, users often find it necessary to add additional filtering conditions or remove some attributes, often based on what they see from the data. In interactive querying, the time to deliver results to the user is most critical. Expensive re-computation that causes much delay (e.g., seconds) for the user severely damages the usability of the system. In this section, we show how our system can dynamically change the representatives according to the new conditions with minimal cost.

4.1 Zoom-in on Representative

When the user sees an interesting tuple from the list of representatives, she can click on the tuple to see more similar items. This operation can be efficiently supported using the cover tree structure. Recall that during the initial medoid generation phase, every final medoid is associated with a CloseSet of nodes in the working level. Those nodes are the set of nodes that is closest to the medoid (relative to all other medoids). Once a medoid s is chosen by the user, we should generate more representatives around s. We proceed as follows. We fetch all nodes in the CloseSet of s, and descend the cover tree to fetch all their children and store them in a list L. This should give us a sample of the nodes/data around medoid s. We treat nodes in list L as the nodes in our new working level. We can run the same algorithm in Sec. 3.3 on nodes in L to obtain a new set of medoids.

4.2 Selection

When a user applies a selection condition, nodes in the working level are very likely to change. As mentioned in Sec. 3.3, existing representatives (medoids) will be eliminated if their CloseSet of nodes are removed by the selection condition. In this section, we detail how to effectively find new medoids when a selection is applied.

First we discuss the effect of a selection condition on each node in the working level. We start with this step because of the procedure we use to generate the medoids. Since the user queries a single table, we can consider a selection condition as a line (in the 2D case) or hyperplane (in 3D or higher dimensionality) in the data universe. For simplicity we now discuss only the 2D case, but high dimensional cases are easy to generalize to. For example, if we use Mileage as x-axis and Price as y-axis in 2D space, adding the selection condition "Price $<$ 12000" removes all data points that are

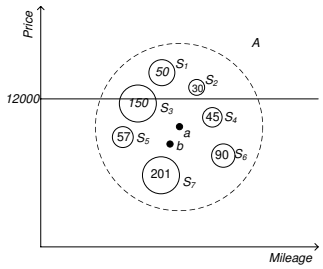


Figure 7: Effect of Selection on a Node

above the line $y = 12000$. Recall that for each node in the cover tree in level i, all its children fall in a circle with radius $D(i)$, and all its descendants are in the circle with radius $2 \times D(i)$ (span). Thus we can determine the impact of a new query condition on a node and its subtree by considering the relationship between the query line and the span of the node. For each node and its subtree can be classified into one of the following categories: 1) completely invalidated, 2) partially invalidated, or 3) completely valid. Category 1 nodes are removed from consideration and category 3 nodes stay intact.

For a category 2 node, once a selection condition is applied and a significant portion of the possible region that a node's descendants can span becomes invalid, the original data point is no longer a suitable approximation of the center of the subtree. The span value of the node is also inaccurate. The example in Fig. 7 shows a category 2 node, node A. The point associated with A is denoted as a, and it is located in the center of the largest circle, which is the span of A. Assume that we have a selection condition "price < 12000 ". For each child s_1 to s_7, the radius of the respective circle denotes the span, and the numerical value denotes the density. After the selection, child nodes s_1 and s_2 are invalid, and s_3 is partially valid.

For partially valid nodes, we use their children to approximate the geometric center of the subtree. Specifically, we treat each child as a point with weight, and calculate the geometric center as the weighted average of all children. We also update the span using the child who is the farthest from the geometric center. Continue with the example in Fig. 7. By averaging over all valid children, we obtain point b as the estimated geometric center of all valid points of the subtree. Now node s_6 is the farthest child from point b. Denote the span of s_6 as $s_6.span$, and the distance from s_6 to b is d. The new span is estimated as the sum of d and $s_6.span$. However, there is a recursion here, since the children can also be partially valid (for example, node s_3). When this happens, we estimate the *valid percentage* of the children as follows. For child node s, in 2D case, we calculate the area around s within distance $s.span$, and calculate the percentage that is still valid under the selection condition. This can be easily extended to higher dimensions. We take into this valid percentage by multiplying it with the node's weight.

After applying the selection condition, if there are less than k valid or partially valid nodes in the working level, we descend the cover tree until a level that has more than k nodes. On the other hand, if we still have more than k nodes in the working level, we can work on the nodes that remain or descend the tree to fetch new nodes. Next, we can re-run the medoid generation algorithm in Sect. 3.3 over the new set of nodes obtained from procedures detailed previously. This gives us a set of updated medoids.

4.3 Projection

We assume the user removes one attribute at a time, which is a reasonable assumption in interactive querying. There is usually some "think time" between two consecutive user actions. Our goal is to refresh the representative without incurring much additional waiting for the user.

Once an attribute is removed, the cover tree index is no longer a valid reflection of the actual distance among data points. Thus the brute-force approach is to re-construct a new cover tree without the attribute and re-compute the medoids. We want to do better than that. Our observations is that although the pair-wise distance between the samples may change dramatically after removing the attribute from consideration, the samples should still represent the data relatively well. Thus we can still use the cover tree as a sampling tool - we sample the data at a level in the cover-tree regardless of the removed attribute.

Our approach is to use the same set of nodes in the working level we used to generate the previous medoids. We remove the dimension chosen by the user. These nodes serve as our primary samples of data. Since the cost is very low to re-run the medoid generation algorithm once we have the seeds, the key is to find a good set of seeds. Using the cover tree as direction is no longer viable: after removing a dimension, nodes that are previously far away can become very close. Also, the weight and span are less accurate in the new distance measure, which may severely affect the quality of generated seeds. So we use Hilbert sort re-order all nodes, and find seeds as outlined in Sec. 3.3. The rest is the same as described in Sec. 3.3.

5. IMPLEMENTATION AND EXPERIMENTS

5.1 System Architecture

The architecture of MusiqLens is shown in Fig. 8. When a query is initially sent from the client user interface to the DBMS, query results are fed to MusiqLens, which interacts with the client in this query session. MusiqLens then builds a cover tree index on the query results. This step can be done very efficiently through cover tree's construction algorithm. One of the features of cover tree is that it can be constructed efficiently in an online fashion. In our experiment, the index for a dataset comprising 130k points in 2D space is built in 0.7 seconds on an Intel Pentium Dual Core 2.8GHz machine with 4GB DDR2 memory. It takes PostgreSQL server 2.7 seconds to output the same set of data (we used a selection without any predicate).

Beside the indexer, the core of MusiqLens contains three other parts: the k-medoid generator, which generates the initial medoids after the user sends a new query to the database; the zooming operator, which is responsible for generating new representatives after user performs a zooming operation; and the query operator, which dynamically adjusts the medoids according to user's new query conditions. MusiqLens can be implemented as a module in a DBMS or a layer between the client and the DBMS. The client interface we built is based on SheetMusiq [17], which is a spreadsheet direct manipulation interface for querying a database.

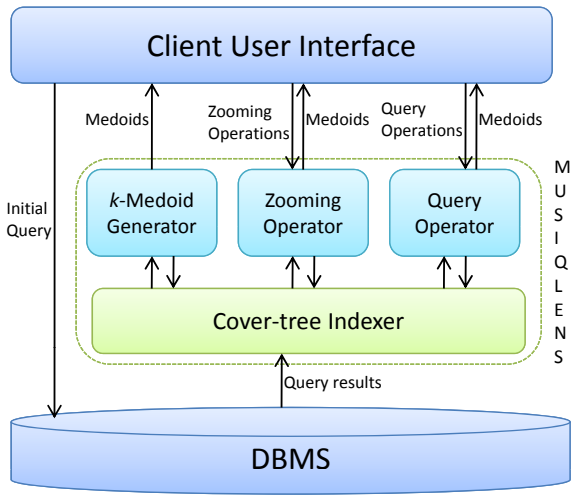


Figure 8: MusiqLens System Architecture

5.2 Experimental Results

The experiments are divided into two parts. First, we want to show that the cover tree based clustering algorithm generates high quality medoids with low time cost. For this we compare our algorithm with R-tree based algorithm in [22], which is the most related work and state-of-the-art. Second, we show that our query-adaptation algorithms work effectively at low time cost and yet do not compromise in quality. We compare algorithms for selection and projection with computing the medoids from scratch.

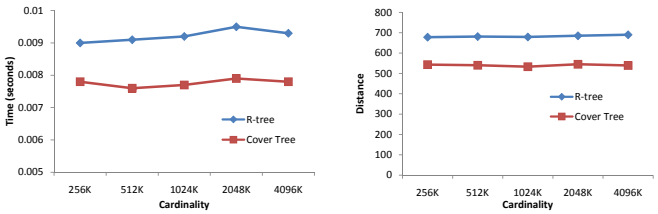
5.2.1 *Comparison with R-tree Based Methods*

We use both real and synthetic data sets for this comparison. We use the LA data set from the R-Tree Portal (http://www.rtreeportal.org). It contains $130k$ million rectangles and we take the center of each rectangle as a data point. We generate synthetic data containing 2-dimensional data points that follow a Zipf distribution with parameter $\alpha = 0.8$. We use 5 sets of data of increasing cardinality: 256K, 512K, 1M, 2M, and 4M. For all data sets, we normalize each dimension of the data to the range of [0, 10000]. We also vary the value of k, the number of medoids to compute. Comparing with R-tree based algorithms, we measure two metrics:

- **Time**: time to compute the medoids
- **Distance**: the average Euclidean distance from data points to their respective medoid.

For convenience, we refer to the cover tree based method as *CTM*, and R-tree based method as *RTM*. In the figures below, legend for CTM is "Cover Tree", and that for RTM is "R-tree".

Fig. 9 shows the time and distance cost with synthetic data sets with growing cardinality, with a fixed value of k at 32. We can see that CTM (Cover Tree based Method) consistently outperforms RTM (R-Tree based Method) in both metrics. Fig. 9 (a) shows that both methods are stable and scalable in time with a growing size of the data set. The primary reason is that, once the R-tree or cover tree index is built, the cost depends more on the value of k (which we will see soon) than on the size of the data. Both algorithms fetch

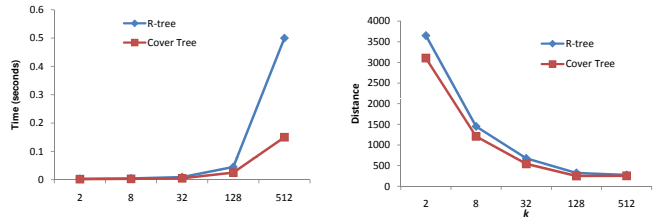


(a) Time (b) Distance

Figure 9: Synthetic Dataset of Various Sizes

the upper levels of the data structure. For a fixed value of k, the number of nodes that need to be read from the disk does not vary significantly with the size of the data. The reason CTM is faster is it brings less data from the disk. Each node of the cover tree is also a data point, and it is smaller than an R-tree node. Fig. 9(b) shows that the distance cost stays stable as the data size varies. This is expected since the medoids are found mainly on the upper levels of the data structure. It also shows that cover tree method produces medoids with much smaller distance cost. In sum, cover tree based method generates better medoids at a lower cost, regardless of the size of the data.

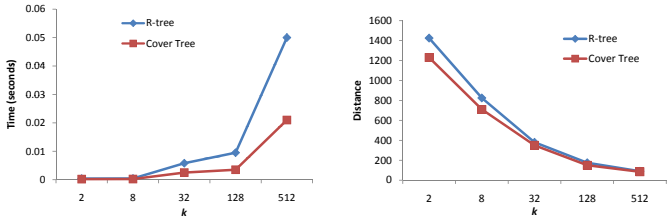
Fig. 10 shows the trend of time and distance cost with the growth of the number of medoids to compute (k), for a synthetic data set with cardinality of $1024k$. We can see that distance from CTM is consistently lower than RTM while using almost half of the time, affirming the conclusion before.



(a) Time (b) Distance

Figure 10: Synthetic Dataset of Various k Values

Fig. 11 shows results on the real data set, LA, with various values of k. The trend in time is the same as we observed in synthetic data, where CTM outperforms RTM by a large margin. CTM also produces better quality medoids than RTM.



(a) Time (b) Distance

Figure 11: Results for Real Dataset

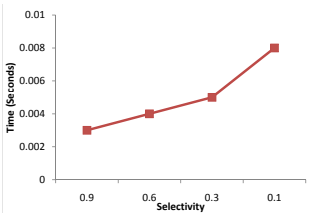
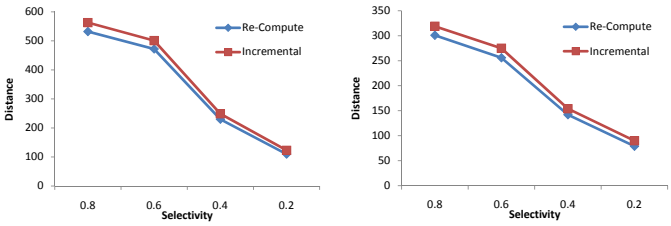


Figure 12: Time for Selection

5.2.2 *Query Refinement*

Having established that cover tree based method is superior than the R-tree based method in generating the initial representatives, we want to see if user issued refinement can be efficiently processed. Since we have no competitor in this incremental re-computation of medoids, we use the absolute running time as the measurement metric. For quality of results, we compare against re-computing the medoids from scratch. For the latter case, when a user issues a selection condition or removes an attribute, we re-build a new cover tree on the data after the query. Thus we are comparing the incremental re-computation algorithms with expensive fresh re-computation.

Selection. We apply selection conditions of various selectivity on a synthetic data set of cardinality $1024k$. The selectivity values are 0.8, 0.6, 0.4, and 0.2. We use selection conditions such as "$x < 4500$" to remove a portion of the data. Since re-constructing a cover tree takes much more time than computing the medoids, there is little meaning to show the time difference. The running time for our algorithm is show in Fig. 12. Since the nodes at the working level can be already cached in the memory when computing previous medoids, we do not need to fetch them from the disk. Possible I/O is still necessary if a large portion of the nodes are disqualified and we need to descend the tree to fetch lower level nodes. The time for selection is well below 0.01 seconds, which is orders of magnitude smaller than re-building the index. In Fig. 13 we show the comparison in distance cost, for both synthetic and real data. The synthetic data is of cardinality $1024k$. We also use the LA data set. We can see that incremental re-computation of medoids using the proposed algorithm (legend "Incremental") provides comparable quality of medoids. There is little, if noticeable at all, difference in terms of distance cost. The time saved is with orders of magnitude, with no compromise of result quality.

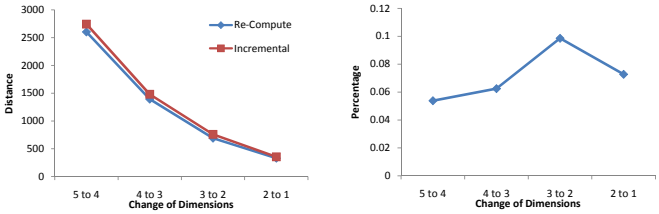


(a) Synthetic Data (b) Real Data

Figure 13: Selection Performance

Projection. We take the same approach as for selection - compare the incremental algorithm with re-computing from scratch. The number of medoids to compute is 32.

We assume that the user projects one attribute at a time. We start with 4 different artificial data sets, each of dimension 5, 4, 3, and 2, respectively. We then remove one dimension from each data set and compare incremental approach with re-computing from scratch. Fig. 14 shows the result. The left figure shows the comparison of absolute distance cost, while the right shows the percentage of increased distance cost using the incremental approach. We can see that the percentage of result compromise is consistently below 10%. In interactive querying, users may not notice the 10% of difference in distance cost, but they will surely notice the difference in time between milliseconds and seconds. Thus we think it is still valuable to save the time of re-building the index and re-computing the medoids at the cost of small deterioration of result quality. Re-computation is the last resort, when the user removes a significant number of dimensions.



(a) Absolute Distance Cost (b) Percentage of Increase

Figure 14: Projection Performance on Single Dimension

5.3 Fast Representative Choice

Cover tree construction is moderately fast – under one second for a moderate size data set (LA) with $131k$ tuples. Fig. 15 shows the time to build a cover tree index for synthetic data sets, with cardinality from 256k up to 4M. We can see that the construction time scales up gracefully. While it is not too expensive from an absolute time perspective, even one second may be too much time to add to how long a user waits to see results.

The encouraging results presented above for incremental computation offer a simple way around this. We precompute the cover tree for the data set – maintenance of this structure is comparable to the cost of incremental index maintenance. Then *every* query against the data set, including the very first, can be treated as a "refinement" of a base query that returns the whole data set. With this, we have an overhead of only 10s of milliseconds per query, a level that is quite affordable.

6. RELATED WORK

Various methods have been proposed for K-medoid clustering. PAM (Partitioning Around Medoids) [15] starts with k randomly selected objects and iteratively improves upon them until the quality of clustering (measured by the average distance to medoids) converges. In each iteration, PAM considers exchanging any of the current k-medoids with any other data point and choosess the swap that leads to the best improvement. This is prohibitively expensive to run on large data sets. CLARA (Clustering LARge Applications)

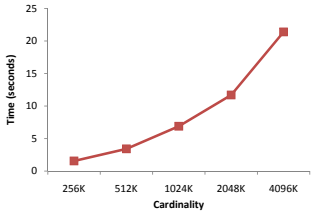


Figure 15: Cover Tree Building Time on Synthetic Data Sets

[15] attempts to reduce the cost by first randomly sampling the data set and then performing PAM on the samples. In order to ensure the samples are sufficiently random, CLARA draws multiple (e.g., 5) sets of samples and uses the best output as the final result. However, in order to estimate the quality of the result, CLARA still needs to compute the distance from all data points to the candidate medoids. This would require scanning the whole data set at least once in each iteration, which is again inefficient for large data sets. Ng and Han's CLARANS (Clustering Large Applications based on RANdomized Search) [23], instead of considering all possible swaps like PAM, randomizes the search and greatly reduces the cost. CLARANS is a main-memory clustering technique, and it also requires scanning the whole data set. For MusiqLens framework, main-memory methods will not suffice since we aim at large data sets.

Some other work uses disk-based indices to speed up the clustering. FOR (focusing on representatives) [6, 8] and TPAQ (tree-based partitioning querying) [21, 22] both assumes that the data set is indexed by an R-tree. FOR takes the most centrally located object out of each leaf node and runs CLARANS on the samples. This means that FOR has to read the entire data set to obtain the samples. TPAQ starts from the root of the R-tree until it reaches a level where there are more than k nodes. It then uses Hilbert curve to sort the nodes and evenly chooses k *seed* nodes out of all nodes in that level. Other nodes are signed to their closest seeds and thus forming small clusters. The geometric center of each cluster is estimated and used to perform a nearest-neighbor (NN) query to fetch the closest point in the data set. The result of each NN query is the medoid of the corresponding cluster. Experiments in [22] shows that TPAQ improves both result quality and computational cost over FOR. FOR and TPAQ are advantageous compared to main-memory methods for the ability to handle large data sets, which is the first requirement of MusiqLens. The second requirement, query adaptation, however, remains unsatisfied by either FOR or TPAQ. For interactive browsing, users may not be so critical on the quality of the medoids, but the lack of interactive refinement and navigation would make the system unusable.

Pan et al [26] proposed an information-theoretic approach for finding representatives from large set of categorical data. They treat each data element as a set of features and obtain a data distribution from the presence of features. It is unclear how to extend the proposed techniques to numerical data, which is the focus of this paper.

Recent work by Li et al [16] proposed generalized group-by and order-by for SQL. Their grid based method is for clustering only, without actually finding the medoid for each cluster. They use a separate ordering function to choose which data point to output for each cluster. To apply techniques in [16] to MusiqLens framework, we would have to find a center for each cluster. One of the methods is to find the point that is closest to the geometric center of the cluster. This would require an additional scan of data or an additional index structure. In addition, [16] does not support zooming, which is an essential feature of MusiqLens. DataScope [32] provides an interface for zooming in and out of a data set by ranking. Common built-in ranking functions are provided (e.g., ranking by the number of publications of authors). The system supports browsing but no searching nor adaption to multiple searching criteria.

Computing k-medoid is related to the problem of clustering. The goal of clustering is to find the naturally close groups in a set of data, where the number of clusters is not known or given a priori. Many efficient techniques have been developed for clustering, for example, BIRCH [34], DBSCAN [7], and CURE [11] (see [33] for a comprehensive survey). The difference between the two problems is, one is to find the natural groupings in the data, and the other is to optimize a distance cost. The cluster centers that naturally exist in the data may not be the best k-medoids, which is shown in [22]. Projection adaptation (Sec. 4.3) is related to the problem of *subspace clustering*, which has been extensively studied [3, 27]. Subspace clustering attempts to find clusters in a data set by selection the most relevant dimensions for each cluster separately. In our case, the set of dimensions to consider are dynamically determined by the user.

Another related problem that has attracted increasing attention is query result diversification [31]. Both [31] and this paper attempt to provide better usability when the number of tuples that can be shown are limited. We believe the two are different solutions under different situations. Diverse results needs ordering of attributes from experts while we do not.

7. CONCLUSION

In this paper, we propose the MusiqLens framework for interactive data querying and browsing, where we solve the "Many-Answers Problem" by showing users representatives of a data set. Our goals are: 1) to find the representatives efficiently, and 2) adapt efficiently when users refine the query. We start by identifying what is a good set of representatives by conducting a user study. Results show consistently that k-medoids are the best amongst seven options. Towards the first goal, we devised cover tree based algorithms for efficiently computing the medoids. Experiments on both real and synthetic data sets shows that our algorithm is superior over our competitor in both time and distance cost. Towards the second goal, we proposed algorithms to efficiently re-generate the representatives when users add selection condition, remove attributes, or zoom-in on frequentatives.

A larger context for the work presented in this paper is the concept of direct manipulation [17, 14, 29], where a user always has in hand data representing some partial result of a search. Thus, someone looking for used cars is shown a few representative cars upon entering the web site, and incrementally refines the result set in multiple steps. The work presented in this paper provides a means for databases to show meaningful results to users without requiring additional effort on their part.

8. ACKNOWLEDGMENTS

This work is supported in part by US NSF grant numbers 0438909 and 0741620. We also thank the anonymous reviewers for their effort and insightful comments.

9. REFERENCES

[1] Database usability research at university of michigan. http://www.eecs.umich.edu/db/usable/.

[2] E. Agichtein, E. Brill, S. T. Dumais, and R. Ragno. Learning user interaction models for predicting web search result preferences. In *SIGIR*, pages 3–10, 2006.

[3] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD Conference*, pages 94–105, 1998.

[4] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML*, pages 97–104, 2006.

[5] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst.*, 31(3):1134–1168, 2006.

[6] M. Ester, H. Kriegel, and X. Xu. *A Database Interface for Clustering in Large Spatial Databases*. Inst. für Informatik, 1995.

[7] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[8] M. Ester, H.-P. Kriegel, and X. Xu. Knowledge discovery in large spatial databases: Focusing techniques for efficient class identification. In *SSD*, pages 67–82, 1995.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[10] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman San Francisco, 1979.

[11] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. In *SIGMOD Conference*, pages 73–84, 1998.

[12] M. Hua, J. Pei, A. W.-C. Fu, X. Lin, and H. fung Leung. Efficiently answering top-k typicality queries on large databases. In *VLDB*, pages 890–901, 2007.

[13] B. J. Jansen and A. Spink. How are we searching the world wide web? a comparison of nine search engine transaction logs. *Inf. Process. Manage.*, 42(1):248–263, 2006.

[14] S. Kandel, A. Paepcke, M. Theobald, H. Garcia-Molina, and E. Abelson. Photospread: a spreadsheet for managing photos. In *CHI*, pages 1749–1758, 2008.

[15] L. Kaufman and P. Rousseeuw. Finding groups in data. an introduction to cluster analysis. *Wiley Series in Probability and Mathematical Statistics. Applied Probability and Statistics, New York: Wiley*, 1990.

[16] C. Li, M. Wang, L. Lim, H. Wang, and K. C.-C. Chang. Supporting ranking and clustering as generalized order-by and group-by. In *SIGMOD Conference*, pages 127–138, 2007.

[17] B. Liu and H. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *ICDE*, 2009.

[18] B. Liu and H. V. Jagadish. Datalens: Making a good first impression. In *SIGMOD Conference, Demonstration Track*, 2009.

[19] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Trans. Knowl. Data Eng.*, 13(1):124–141, 2001.

[20] F. Morii. A generalized k-means algorithm with semi-supervised weight coefficients. In *ICPR (3)*, pages 198–201, 2006.

[21] K. Mouratidis, D. Papadias, and S. Papadimitriou. Medoid queries in large spatial databases. In *SSTD*, pages 55–72, 2005.

[22] K. Mouratidis, D. Papadias, and S. Papadimitriou. Tree-based partition querying: a methodology for computing medoids in large spatial datasets. *VLDB J.*, 17(4):923–945, 2008.

[23] R. T. Ng and J. Han. Clarans: A method for clustering objects for spatial data mining. *IEEE Trans. on Knowl. and Data Eng.*, 14(5):1003–1016, 2002.

[24] R. Nosofsky and S. Zaki. Exemplar and prototype models revisited: Response strategies, selective attention, and stimulus generalization. *Learning, Memory*, 28(5):924–940, 2002.

[25] C. R. Palmer and C. Faloutsos. Density biased sampling: An improved method for data mining and clustering. In *SIGMOD Conference*, pages 82–92, 2000.

[26] F. Pan, W. Wang, A. K. H. Tung, and J. Yang. Finding representative set from massive data. In *ICDM*, pages 338–345, 2005.

[27] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data: a review. *ACM SIGKDD Explorations Newsletter*, 6(1):90–105, 2004.

[28] H. Shin and R. Nosofsky. Similarity-scaling studies of dot-pattern classification and recognition. *Journal of Experimental Psychology: General*, 121(3):278–304, 1992.

[29] B. Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[30] J. Smith and J. Minda. Prototypes in the mist: The early epochs of category learning. *Learning, Memory*, 24(6):1411–1436, 1998.

[31] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia. Efficient computation of diverse query results. In *ICDE*, pages 228–236, 2008.

[32] T. Wu, X. Li, D. Xin, J. Han, J. Lee, and R. Redder. Datascope: Viewing database contents in google maps' way. In *VLDB*, pages 1314–1317, 2007.

[33] R. Xu and I. Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645, 2005.

[34] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *SIGMOD Conference*, pages 103–114, 1996.