Addressing Diverse User Preferences
in SQL-Query-Result Navigation

Zhiyuan Chen
University of Maryland, Baltimore County
Baltimore, NY, USA
zhchen@umbc.edu

Tao Li *
Florida International University
Miami, FL, USA
taoli@cs.fiu.edu

ABSTRACT

Database queries are often exploratory and users often find their queries return too many answers, many of them irrelevant. Existing work either categorizes or ranks the results to help users locate interesting results. The success of both approaches depends on the utilization of user preferences. However, most existing work assumes that all users have the same user preferences, but in real life different users often have different preferences. This paper proposes a two-step solution to address the diversity issue of user preferences for the categorization approach. The proposed solution does not require explicit user involvement. The first step analyzes query history of all users in the system offline and generates a set of clusters over the data, each corresponding to one type of user preferences. When user asks a query, the second step presents to the user a navigational tree over clusters generated in the first step such that the user can easily select the subset of clusters matching his needs. The user then can browse, rank, or categorize the results in selected clusters. The navigational tree is automatically constructed using a cost-based algorithm which considers the cost of visiting both intermediate nodes and leaf nodes in the tree. An empirical study demonstrates the benefits of our approach.

Categories and Subject Descriptors

H.2 [**Information Systems**]: DATABASE MANAGEMENT; H.3.3 [**Information Systems**]: INFORMATION STORAGE AND RETRIEVAL—*Information Search and Retrieval*

General Terms

Algorithms, Human Factors

Keywords

User Preferences, Data Exploration

1. INTRODUCTION

As internet becomes ubiquitous, many people are searching their favorite cars, houses, movies, stocks, etc. over the web. Many websites use databases to store the data and provide a form-based search interface. However, typically users often can not form a query that returns exactly the answers matching their preferences. Instead, they will start with some queries with very general conditions and returning many answers, and then iteratively refine their queries until a few answers matching their preferences are returned. However, this iterative procedure is time-consuming and many users will give up before they reach the final stage.

To speed up this iterative search process, two types of solutions have been proposed. The first type [7] categorizes the query results into a *navigational tree*, and the second type [2, 9, 10, 1, 8] ranks the results. The success of both approaches depends on the utilization of user preferences. However, most existing work assumes that all users have the same user preferences, but in real life different users often have different preferences.

Example 1. Consider a mutual fund selection website. Figure 1 shows a fraction of a navigational tree generated using a method proposed in [7] over 193 mutual funds (details are described in Section 6.1) returned by a query with the condition fund_name like '%Vanguard%'. Each tree node specifies the range or equality conditions on an attribute, and the number in the parentheses is the number of data records satisfying all conditions from the root to the current node. Users can use this tree to select the funds that are interesting to them. For example, a user interested in funds with very high returns may select those funds with "3 Yr return" over 20% (the right most node in the first level). Now he only needs to examine 29 records instead of the 193 records in the query results.

Consider four users U1, U2, U3, and U4. U1 and U2 prefer funds with high returns and are willing to take higher risks, U3 prefers funds with low risks and is willing to sacrifice some returns, and U4 prefers both high return funds and low risk funds (these two types of funds typically do not overlap). The existing method assumes that all users have the same preferences. Thus it places attributes "3 Year return" and "1 Year return" at the first two levels of the tree because more users are concerned with returns than risks. However, the attribute characterizing the risks of a fund is "standard deviation", and is placed at multiple nodes in the third level of the tree. Suppose U3 and U4 want to visit low risk funds at the two bottom-left nodes (bounded with rectangles) in Figure 1, they need to visit many nodes (including nodes on the paths from the root plus the sibling nodes because users need to check the labels of the siblings to decide which path to follow.).

Two key challenges: User preferences are often difficult to obtain because users do not want to spend extra efforts to specify their preferences. Thus there are two major challenges to address the
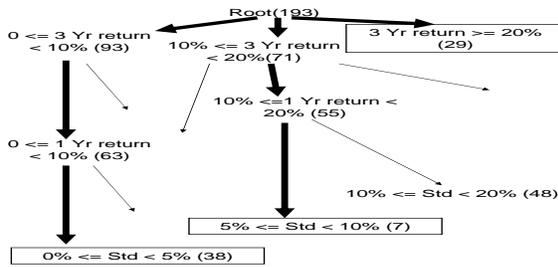
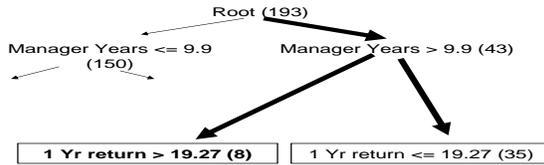**Figure 1: Tree generated by existing method**

Figure 2: Tree generated by our method

diversity issue of user preferences: (1) how to summarize diverse user preferences from the behavior of all users already in the system, and (2) how to decide the subset of user preferences associated with a specific user. As most existing work [7, 2, 9, 10] did, we use query history to infer the behavior of all users in the system.

Agrawal et al. [1] addressed the problem of diverse user preferences for the ranking approach. The authors assume there are a set of orders over records, each corresponding to some contexts (represented as conditions over attributes). A method is proposed to select a small subset of representing orders. When a user asks a query, the query will be matched against the contexts. An order will be generated to maximize the agreement of stored orders with the returned order, where each stored order will be weighted with the similarity score between the query and the context associated with that order. However, this work does not consider the case of categorization. Further, it does not focus on how to summarize user preferences from the behavior of all users (it assumes preferences are given). It also uses the query asked by a specific user to decide his preferences, which is questionable because as mentioned earlier, typically users can not form a meaningful query at the beginning of their searches.

One well-known solution to the second challenge is to define a user profile for each user and use the profile to decide his preferences. However, in real life user profiles may not be available because users do not want to or can not specify their preferences (if they can, then they can form the appropriate query and there is no need for either ranking or categorization). One could try to derive a profile from the query history of a certain user, but this method will not work if the user is new to the system, which is exactly when the user needs help the most.

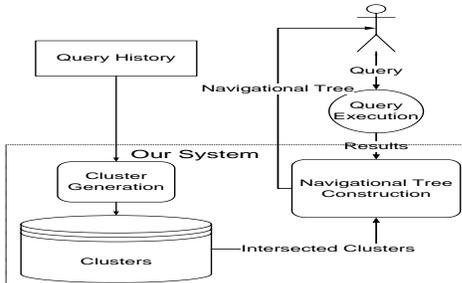**Our approach:** In this paper, we propose a two-step approach to

Figure 3: System architecture.

address both challenges for the categorization case. The system architecture is shown in Figure 3. The first step occurs offline. It analyzes query history of all users already in the system and generates a set of non overlapping clusters over the data, each corresponding to one type of user preferences. Each cluster has an associated probability of users being interested in that cluster. We assume that an individual user's preference can be represented as a subset of these clusters, and each cluster will appear in the subset with its associated probability. The system stores these clusters (by adding a class label to each data record) and probabilities for each cluster.

The second step occurs online when a specific user asks a query. It first intersects the set of clusters generated in the first step with the answers of the query. It then automatically constructs a navigational tree over these intersected clusters on the fly. This tree is then presented to the user. The user first navigates this tree to select the subset of clusters matching his needs. The user then can browse, rank, or categorize the results in the selected clusters.

Note that here we are not intending to select the most interesting records for the user. Instead, it is up to the user to do so. We just provide a navigational tree that "best describes" the differences of different clusters such that the user can easily locate the subset of clusters matching his needs. This step also does not assume the availability of a user profile or a meaningful query.

The diversity issue is addressed in two aspects. First, the first step of our approach captures diverse user preferences by identifying all types of user preferences in the format of clusters. Second, the second step uses a navigational tree to let a specific user select the subset of clusters (or types of user preferences) matching his needs. Let k be the number of clusters, the user can potentially select up to 2^k different preferences (subsets). This gives users more flexibility than the profiling approach because in the latter case each user is forced to use only one profile.

In Example 1, suppose the first step of our approach generates three clusters, one for high return funds, one for low risk funds, and the third for the remaining funds (i.e., those no users will be interested in).

In the second step, when a user asks for "Vanguard mutual funds", a navigational tree shown in Figure 2 is automatically constructed. How this tree is generated will be discussed later. Here we just give some intuitions. Attribute "manager years" is selected in the first level of the tree. The intuition is that funds managed by very senior managers often have either very high returns, or very low risks because otherwise those managers may get fired for their poor performance. "manager years" is not selected by the existing categorization method [7] because the most frequently used attribute is "3 Yr return" in the query history. Next "1 year return" is selected to separate the remaining two clusters. Now "high return" funds and "low risk" funds can be found at two leaf nodes with bounding rectangles. All four users only need to visit 4 tree nodes (2 in first level and 2 in the second level) excluding the root, while using the tree in Figure 1, U3 and U4 need to visit 12 nodes (3 in the first level, 5 in the second level, and 4 in the third level).

Our contributions are summarized as follows:

- We propose a clustering method to summarize user preferences of all users in the system using query history. This method uses query pruning, query merging, precomputation, and min-wise hashing to deal with large query histories and large data sets.

- We propose a cost-based algorithm to construct a navigational tree over these clusters. Unlike existing categorization and decision tree construction algorithms, this algorithm considers the overhead for users to visit both intermediate nodes and leaf nodes.

- We conduct an empirical study which demonstrates the superiority of our approach over existing solutions.

Road map: The rest of the paper is organized as follows. Section 2 describes related work. Section 3 gives necessary preliminaries. Section 4 describes the clustering step of our approach. Section 5 describes the tree construction step. Section 6 presents results of an empirical study. Section 7 concludes the paper.

2. RELATED WORK

The only known study on categorizing SQL query results [7] proposed a greedy algorithm to construct a navigational tree. This algorithm uses query history of all users in the system to infer an overall user preference as the probabilities of users are interested in each attribute. As explained in Section 1, it does not consider the diversity issue of user preferences. Further, we will show in Section 5 that the algorithm does not consider the impact of future splits on the overhead of navigating the tree.

There has been a rich body of work on categorizing text documents [12, 16, 18] and web search results [28, 27]. However, categorizing relational data presents unique challenges and opportunities. First, relational data contains numerical values while text categorization methods treat documents as bags of words. Further, this paper tries to minimize the overhead for users to navigate the generated tree (will be defined in Section 3), which is not considered by existing text categorization methods.

There has been a rich body of work on information visualization [6] techniques. Two popular techniques are dynamic query slider [3] which allows users to visualize dynamic query results using sliders to represent range search conditions, and brushing histogram [26] which employs interactive histograms to represent each attribute and helps users exploring correlations between attributes. However, none of them takes query history into account. Furthermore, information visualization techniques also require users to specify what information to visualize (e.g., by setting the slider or selecting histogram buckets). Since our approach generates the information to visualize, i.e., the navigational tree, our approach is complementary to visualization techniques. For example, if the leaf of the tree still contains many records, a query slider or brushing histogram could be used to further narrow down the scope.

There has been recent work on applying ranking techniques for data stored in databases [2, 9, 10, 1, 8]. Ranking is complementary to categorization and we could use ranking in addition to our techniques (e.g., we could rank records stored in each leaf). Further, most existing work does not consider the diversity issue of user preferences. The only exception is [1]. However it does not focus on how to obtain different user preferences. To the best of our knowledge, our paper is the first to consider the diversity issue of user preferences for the categorization approach.

There also has been a lot of work on information retrieval [22, 13, 23, 17] using query history or other implicit feedbacks. However, such work focuses on searching text documents, while this paper focuses on searching relational data. These studies also typically rank query results, while this paper categorizes the results.

One could also use existing hierarchical clustering techniques [19] to create the navigational tree. However, the trees generated are not easy for users to navigate. For example, how do we describe the records contained in a node? We can use a representative record, but such a record may contain many attributes, making it difficult for users to read. On the contrary, the navigational tree is easy to understand because each node just uses one attribute.

There has also been work on OLAP queries [15] that aggregate data into a hierarchical structure and allow users to drill down or

Table 1: Symbols

D	Dataset
H	Query history
T	Navigational tree
A_i	Attribute i
Q_i	Query i
F_i	Frequency of query i
D_{Qi}	Results of Q_i
C_i	Preference-based cluster i
P_i	Probability of users being interested in C_i
QC_i	Query cluster i
t	Tree node
$Anc(t)$	The set of ancestors of node t
$Sib(t)$	The set of siblings of node t
$N(t)$	Number of records in node t

roll up. However, users have to specify the group by attributes while in this paper the tree is created automatically.

3. PRELIMINARIES

Notations: Table 1 lists the symbols used in this paper. Let D be a relation with n records $r_1, ..., r_n$ and m attributes $A_1, ..., A_m$. Let H be a query history $\{(Q_1, U_1, F_1), ..., (Q_k, U_k, F_k)\}$ in chronological order, where Q_i is a query, U_i is a session ID (a session starts when a user connects to the database and ends when the user disconnects), and F_i is a weight associated with that query. In this paper, F_i represents the frequency of the query. But the methods proposed in this paper also apply to other meanings of F_i as well (e.g., the importance of the query). Note that queries in the same session are asked by the same user, which will be used later to prune queries. The query history can be collected easily using the query logging feature of commercial database systems.

We assume that all queries only contain range or equality conditions, and the conditions can be represented in the following conjunctive normal form: $Cond(A_1) \wedge Cond(A_2) \ldots \wedge Cond(A_m)$. $Cond(A_i)$ is the conditions on attribute A_i and is in the format of $c_{11} \leq A_i \leq c_{12} \vee c_{21} \leq A_i \leq c_{22} \vee \ldots \vee c_{1j} \leq A_i \leq c_{2j}$, where $c_{11}, c_{12}, \ldots, c_{2j}$ are constants.

D can be partitioned into a set of disjoint *preference-based clusters* $C = \{C_1, ..., C_q\}$, where each cluster C_j corresponds to one type of user preferences. Each C_j is associated with a probability P_j that users are interested in C_j. This set of clusters are inferred from the query history.

We assume that the clusters are disjoint in this paper just for simplicity. In practice, user preferences may overlap. However, we may divide two overlapped preferences into three clusters, one for the intersection part of the two preferences, and the other two for the non-overlapped parts of each preference. We also assume that each user's preference may consist of a set of clusters instead of just one cluster. Thus although the clusters are not overlapped, our model can represent overlapped user preferences.

We also assume that the results of queries are not stored in the query history, and can be recomputed later. We further assume that the dataset D is fixed. In practice D may get modified from time to time. In this case we assume that clusters are generated periodically (e.g., once a month), and the result D_{Q_i} of a query $Q_i \in H$ is computed using the database at the time when clusters are generated. This is because users are typically more interested in the most up-to-date information. For example, the "1 Year Return" of a mutual fund may get updated frequently. Thus it makes more sense to use the current value rather than using previous ones.

Definition of navigational tree and navigational cost:

DEFINITION 1. *A navigational tree $T(V, E, L)$ contains a node set V, an edge set E, and a label set L. Each node $v \in V$ has a label $l(v) \in L$ which specifies the condition on an attribute such that the following should be satisfied: (1) such conditions are range or equality conditions, and the bounds in the range conditions are called split points; (2) v contains records that satisfy all conditions on its ancestors including itself; (3) conditions associated with children of a non-leaf node v are on the same attribute (called split attribute), and define a partition of the records in v.*

The third condition in Definition 1 restricts that each node only uses one split attribute. This makes it simple for users to decide which branch to follow.

We assume that a user visits T in a top-down fashion, and stops at a leaf that contains the records that he is interested in. Let t be a leaf of T with $N(t)$ records and C_j be a cluster in C. $C_j \cap t \neq \emptyset$ denotes that t contains records in C_j. $Anc(t)$ denotes the set of ancestors of t including t itself, but excluding the root. $Sib(t)$ denotes the set of sibling nodes of node t including itself. Let w_1 and w_2 represent the weights (or unit costs) of visiting a record in the leaf and visiting an intermediate tree node, respectively. The navigational cost is defined as follows.

DEFINITION 2. *The navigational cost $NCost(T, C)$ equals*

$$\sum_{t \in Leaf(T)} \sum_{C_j \cap t \neq \emptyset} P_j(w_1 N(t) + w_2 \sum_{t_i \in Anc(t)} |Sib(t_i)|).$$

The navigational cost of a leaf t consists of two terms: the cost of visiting records in leaf t and the cost of visiting intermediate nodes. Users also need to examine the labels of all sibling nodes to select a node on the path from the root to t. Thus users have to visit $\sum_{t_i \in Anc(t)} |Sib(t_i)|$ intermediate tree nodes. The cost of visiting the root is excluded because every tree has a root. When users reach the leaf t, they have to look at all $N(t)$ records in t. For example, in Figure 1, suppose a user is interested in the node with the left most rectangle with 100% probability, the user needs to visit all three nodes in the first level, the left two nodes in the second level, the left two nodes in the third level, and 38 records in that leaf. Let $w_1 = w_2 = 1$, the total cost equals $3 + 2 + 2 + 38 = 45$.

Let P_j be the probability that users will be interested in cluster C_j. Definition 2 computes the expected cost over all clusters and leaves.

Next we define two problems: (1) the problem of summarizing preferences of all users via clustering, and (2) the problem of finding an optimal navigational tree for a query.

The clustering problem: A co-clustering method was proposed in [11] to cluster documents and keywords. We could apply this method to co-cluster data and queries in the query history. However, each query is assigned to just one cluster. In our settings, the same query may belong to multiple clusters of records. For example, funds with high returns and funds with low risks may both be managed by senior managers, thus the query asking for funds managed by senior managers may appear in both clusters.

We generate preference-based clusters as follows. We first define a binary relationship R over records such that $(r_i, r_j) \in R$ if and only if two records r_i and r_j appear in the results of the exactly same set of queries in H. If $(r_i, r_j) \in R$, according to the query history, r_i and r_j are not distinguishable because each user that requests r_i also requests r_j and vice versa. Clearly, R is reflexive, symmetric, and transitive. Thus R is an equivalence relation and it partitions D into equivalence classes $\{C_1, \ldots, C_q\}$, where records equivalent to each other are put into the same class. Those records not selected by any query will also form a cluster associated with zero probability (since no users are interested in them).

```
CreateTree(H,D, Q)
1.  (This step occurs offline and only once.)
Cluster records in query results D using H.
The results are a set of clusters C_1,...,C_q,
and each cluster C_j,1 ≤ j ≤ q, has a probability P_j.
2.  (This step occurs online for each query Q).
Create a navigational tree T with minimal
NCost(T,C) over records in results of Q,
using C_1,...,C_q as class labels.
```
Figure 4: Algorithm 1.

PROBLEM 1. *Given database D, query history H, find a set of disjoint clusters $C = \{C_1, \ldots, C_q\}$ such that for any records r_i and $r_j \in C_l, 1 \leq l \leq q$, $(r_i, r_j) \in R$, and for any records r_i and r_j not in the same cluster, $(r_i, r_j) \notin R$.*

Example 2: Suppose there are three queries Q_1, Q_2, and Q_3 and 13 records $r_1, r_2, \ldots, r_{13}$. Q_1 returns first 10 records $r_1, r_2, \ldots, r_{10}$, Q_2 returns the first 9 records $r_1, r_2, \ldots r_9$, and r_{11}, and Q_3 returns r_{12}. Clearly the first 9 records $r_1, r_2, \ldots r_9$ are equivalent to each other since they are returned by both Q_1 and Q_2. The data can be divided into five clusters $\{r_1, r_2, \ldots, r_9\}$ (returned by Q_1, Q_2), $\{r_{10}\}$ (returned by Q_1 only), $\{r_{11}\}$ (returned by Q_2 only), $\{r_{12}\}$ (returned by Q_3), and $\{r_{13}\}$ (not returned by any query).

If we compute for each record r_i, the set of queries that contain it in the results, a simple grouping operation can generate the clusters. However, in practice, the query history H may contain many queries, thus the direct application of the above method would generate too many clusters. This will make it difficult for a user to select his preferred clusters in the second step. Further, the dataset D may also contain many records, making it expensive to generate the clusters. Thus we propose several techniques to address this problem in Section 4.

The navigational tree construction problem:

PROBLEM 2. *Given D, C, Q, find a tree $T(V, E, L)$ such that (1) it contains all records in the results of Q, (2) there does not exist another tree T' satisfying (1) and with $NCost(T', C) < NCost(T, C)$.*

The above problem can be proved to be NP hard in a way similar to proving that the problem of finding an optimal decision tree with a certain average length is NP hard. Section 5 will present an approximate solution.

Figure 4 shows our approach. Next we describe the clustering step in Section 4 and the tree construction step in Section 5.

4. PREFERENCE-BASED CLUSTERING

This section describes the algorithm to cluster data records using query history. We propose two preprocessing steps to prune unimportant queries and merge similar queries. The algorithm is described in Figure 5. At step 1 and 2, the algorithm prunes unimportant queries and merges similar queries into a single query. At step 3 to 5, the clusters are generated.

Query pruning: The pruning algorithm is based on the following heuristics: (1) queries with empty answers are not useful, (2) in the same session, a user often starts with a query with general conditions and returns many answers, and then continuously refines the previous query until the query returns a few interesting answers. Therefore, only the last query in such a refinement sequence is important. We define a refinement relationship denoted as $\subseteq$ between queries as follows.

DEFINITION 3. *Let D_{Q_i} and D_{Q_j} be results of Q_i and Q_i. $Q_i \subseteq Q_j$ if and only if $D_{Q_i} \subseteq D_{Q_j}$.*

An obvious way to verify the refinement relationship is to execute a SQL statement as Q_i minus Q_j and check if the result

```
Cluster(H, D_Q)
1.   H'=Prune(H, D).
2.   {QC_1,...,QC_k} = Merge(H')
3.   For each record r_i ∈ D_Q, identify
S_i = {QC_p|∃Q_j ∈ QC_p such that r_i is returned
by Q_j}.
4.   Group records in D_Q by S_i.
5.   Output each group as a cluster C_j,
assign a class label for each cluster,
and compute probability P_j = (Σ_{Q_i∈S_j} F_i)/(Σ_{Q_p∈H'} F_p).
```

Figure 5: The algorithm to generate preference-based clusters.

```
Prune(H,D)
1. Prune queries with empty answers.
2. Order the remaining queries by
session ID and then by chronological order.
3. For each longest sequence (Q_i,U_i,F_i),
...,(Q_j,U_j,F_j) in H
such that U_i = U_{i+1} = ... = U_j and
Q_i ⊆ Q_{i+1}... ⊆ Q_j and (Q_j ⊈ Q_{j+1} or U_j ≠ U_{j+1}).
4.   Prune all queries Q_i,...,Q_{j-1}.
5. End For
```

Figure 6: The query pruning algorithm.

is empty. Since this paper only considers queries with range and equality conditions, we can use a more efficient algorithm described below which does not require execution of any query.

Let Q_i's condition on attribute A_l be $c_{11} \leq A_l \leq c_{12} \vee c_{21} \leq A_l \leq c_{22} \vee \ldots \vee c_{1p} \leq A_l \leq c_{2p}$. and Q_j's condition on attribute A_l be $c'_{11} \leq A_l \leq c'_{12} \vee c'_{21} \leq A_l \leq c'_{22} \vee \ldots \vee c'_{1q} \leq A_l \leq c'_{2q}$. $Q_i \subseteq Q_j$ if for every condition $c_{1x} \leq A_l \leq c_{2x}$ in Q_i, Q_j either does not contain any condition on A_l or has a condition $c'_{1y} \leq A_l \leq c'_{2y}$ such that $c'_{1y} \leq c_{1x} \leq c_{2x} \leq c'_{2y}$. For example, if Q_1 has condition "1 year return > 0", and Q_2 has condition "1 year return > 10%", then $Q_2 \subseteq Q_1$. Note that this algorithm actually checks a stronger form of refinement which does not depend on the dataset D (or the refinement must hold for any D). This may lead to fewer queries being pruned, but is more efficient than executing queries. Figure 6 shows the query pruning algorithm.

Query merging: We could certainly merge queries based on semantic similarities. However, since we will use queries to partition data into clusters, a more meaningful method is to measure the similarity between the set of answers returned by two different queries. Let D_{Qi} and D_{Qj} be the set of records returned by Q_i and Q_j. Below is the definition of distance between Q_i and Q_j.

DEFINITION 4. *The distance $d(Q_i, Q_j)$ between Q_i and Q_j equals $1 - \frac{|D_{Qi} \cap D_{Qj}|}{|D_{Qi} \cup D_{Qj}|}$.*

$\frac{|D_{Qi} \cap D_{Qj}|}{|D_{Qi} \cup D_{Qj}|}$ is also called *resemblance* between D_{Qi} and D_{Qj}. In Example 2, the intersection of answers to Q_1 and Q_2 consists of the first 9 records, and the union of answers consists of 11 records. Thus the distance between Q_1 and Q_2 equals $1 - 9/11 = 0.18$. The query merging problem is defined as follows.

PROBLEM 3. *Given H' (the pruned query history), D, and a distance bound B, find the minimal number k such that queries in H' can be divided into k disjoint clusters $QC_1, \ldots, QC_k$ where for any $Q_i, Q_j \in QC_l, 1 \leq l \leq k, d(Q_i, Q_j) \leq B$.*

The distance bound B is a user defined parameter and we set it to 0.2 in this paper because in experiments this value greatly reduces the number of queries and at the same time only merges very similar queries.

Problem 3 is NP hard because a simplification of this problem, deciding whether a set of data points can be divided into 3 clusters,

```
Merge(H',D, B)
1. Compute result D_{Q_i} for each query Q_i ∈ H'
2. Create a cluster QC_i for each query Q_i ∈ H'
3. For every pair of QC_i and QC_j
 compute the average distance between
 pairs of queries in QC_i and QC_j
4. Merge the pair of QC_i and QC_j
with the smallest average distance
if for all pairs of queries in QC_i and QC_j,
the distance is below B.  Delete both
clusters, add a new cluster as QC_i ∪ QC_j
5. Repeat step 3 and 4 until no more
merging is possible
6. Return the merged query clusters.
```

Figure 7: The query merging algorithm.

is NP hard. Thus we propose a Greedy algorithm in Figure 7. We also assume that the result set D_{Q_i} consists of the record IDs of records returned by Q_i.

This algorithm repeatedly merges pairs of query clusters and stops when no pairs can be merged. Note that the database is only accessed in line 1 of the algorithm where we compute the result sets of all queries. Later we only need to use the result sets consisting of record IDs. We can further limit the I/O cost by scanning the entire database just once. During this scan, for each record in the database, we evaluate the conditions in each query against this record to decide whether the query returns that record.

Let the time to compute distance be t_d, and the complexity of this algorithm be $O(|H'||D| + |H'|^3 t_d)$. The first term is the complexity to evaluate $|H'|$ queries with range or equality conditions on a size $|D|$ database (line 1 in Figure 7). The second term is the complexity for merging operations (line 2-6 in Figure 7). For each merge operation the algorithm needs to compute the distance between $O(|H'|^2)$ pairs of queries and there can be at most $|H'|$ merges. t_d is $O(|D|)$ because in the worst case the sizes of D_{Qi} and D_{Qj} are $|D|$. Thus this algorithm does not scale for large data sets and large workloads. Next we describe several techniques to speed up the merging algorithm.

Sampling: The first technique generates a random sample from the query history if the size of the history is very large. We first randomly select a set of users, and then select queries asked by these users. We do not sample on the query level because the queries asked by the same user are often related.

Precomputation: The second technique avoids redundant distance computation by precomputing the distances between all pairs of queries at line 1 of the algorithm, and only keeps those pairs whose distance is below B. Those pairs of queries not retained will never be merged. Further, at line 3-4 of the algorithm, we speed up the selection of the pair of clusters with minimal distance by using a binary heap to maintain the distances for all pairs of clusters.

The size of the heap is $O(|H'|^2)$ and the heap can be created in $O(|H'|^2 \log |H'|^2) = (|H'|^2 2 \log |H'|) = O(|H'|^2 \log |H'|)$ time. Now for each merge operation (line 3-4 of the algorithm), it takes $O(1)$ time to extract the pair with minimal distance. Next we examine how to maintain the heap.

Suppose the pair of QC_i and QC_j is merged at line 4 of the algorithm, it is easy to verify that only the distances of those pairs of clusters involving either QC_i or QC_j need to be updated. For example, if a pair consists of QC_i and QC_p, then we need to update the average distance of this pair to the average distance between $QC_i \cup QC_j$ and QC_p. There can be at most $O(|H'|)$ pairs involving QC_i or QC_j because the total number of clusters does not exceed $|H'|$. Moreover, let $TD(QC_i, QC_p)$ be the total distance

between queries in QC_i and QC_p. We have:

$$TD(QC_i \cup QC_j, QC_p) = TD(QC_i, QC_p) + TD(QC_j, QC_p)$$

Thus the distance between $QC_i \cup QC_j$ and QC_p can be inferred from the distances for (QC_i, QC_p) and (QC_j, QC_p) in $O(1)$ time. Updating a heap node takes $O(\log(|H'|^2)) = O(\log |H'|)$ time. Each merging step (line 3 and 4 of the algorithm) will take $O(|H'| \log |H'|)$ time because we need to update $O(|H'|)$ nodes in the heap. The complexity of the algorithm is reduced to $O(|H'||D| + |H'|^2 t_d + |H'|^2 \log |H'|)$, where the second term is the precomputation cost and the third term is the cost of merging operations.
Min-wise hashing: The third technique uses min-wise hashing [5] to speed up distance computation for large data sets. The key idea of min-wise hashing is that we can create a small signature (e.g., with 100 integers) for the result set D_{Qi} of each query Q_i, and the resemblance of any pair of D_{Qi} and D_{Qj} can be estimated accurately using these signatures.

The min-wise hashing signature is computed as follows. Given a signature size l, we first generate l independent random hash functions $f_1, \ldots, f_l$. For result set D_{Qi}, the p'th component of its signature equals

$$\min_{r \in D_{Qi}} f_p(r)$$

That is, we record the minimal hash value of all record IDs in D_{Qi} for the p'th hash function f_p. Note that the same hash function f_p is used for every query to generate its p'th signature component. Let the signature of D_{Qi} and D_{Qj} be $Sig(Q_i)$ and $Sig(Q_j)$, respectively. Let $Sig(Q_i)_p$ be the p'th component of the signature. We call it a match if $Sig(Q_i)_p = Sig(Q_j)_p$. The resemblance of D_{Qi} and D_{Qj} can be estimated by the number of matches of $Sig(Q_i)$ and $Sig(Q_j)$ divided by signature size l. Thus we can estimate the distance using these signatures because the distance between Q_i and Q_j equals one minus the resemblance of D_{Qi} and D_{Qj}. In Example 2, suppose the signature for Q_1 is $(1, 9, 8, 4, 10)$ and the signature for Q_2 is $(1, 9, 8, 4, 11)$, then there are 4 matches and the resemblance of D_{Q1} and D_{Q2} is estimated as $4/5 = 0.8$.

The min-wise hashing estimator is unbiased. An error bound was given in [5] and the accuracy increases with the resemblance value. This suits our needs perfectly because we only merge queries with high resemblance. The accuracy also increases with the size of signatures. We use signature size of 100 in this paper because it gives high accuracy.

Signatures of a query can be computed in $O(|D|)$ time, and the number of matches between two signatures can be computed in $O(l)$ time. Note that l is irrelevant to the size of database. Thus the complexity of merging becomes $O(|H'||D| + |H'|^2 l + |H'|^2 \log |H'|)$, using both min-wise hashing and precomputation of distances. As mentioned before, this algorithm also only scans the whole database just once. Further, the clustering step occurs offline and only needs to be done once.
Generation of clusters: After query pruning and merging, we get a set of query clusters $QC_1, \ldots, QC_k$. For each record r_i, we generate a set S_i consisting of query clusters such that one of the queries in that cluster returns r_i. That is, $S_i = \{QC_p | \exists Q_j \in QC_p$ such that r_i is returned by $Q_j\}$. We then group data records according to their S_i, and each group forms a cluster. Each cluster is assigned a class label. The probability of users being interested in cluster C_i is computed as the sum of probabilities that a user asks a query in S_i. This equals the sum of frequencies of queries in S_i divided by the sum of frequencies of all queries in the pruned query history H'. In example 2, after merging we get two clusters $\{Q1, Q2\}$ and $\{Q3\}$. Three clusters C_1, C_2, and C_3 will be generated. C_1 corresponds to Q_1 and Q_2 and contains the first 11 records, with probability $P_1 = 2/3 = 0.67$. C_2 corresponds to Q_3 and contains r_{12}, with

probability $P_2 = 1/3 = 0.33$. C_3 contains r_{13}, with probability 0 because r_{13} is not returned by any query.

5. NAVIGATIONAL TREE CONSTRUCTION

This section describes the navigational tree construction algorithm. Section 5.1 explains the relationship of navigational tree construction to decision tree construction and gives an overview of our algorithm. Section 5.2 proposes a novel splitting criterion that considers the cost of visiting both leaves and intermediate nodes.

5.1 Algorithm Overview

A navigational tree is very similar to a decision tree. There are many well-known decision tree construction algorithms such as ID3 [20], C4.5 [21], and CART [4]. There have also been studies on cost-sensitive splitting criteria [24, 25]. However, the existing decision tree construction algorithm aims at minimizing the impurity of data [21] (represented by information gain, etc.). Our goal is to minimize the navigational cost, which includes both the cost of visiting intermediate tree nodes and the cost of visiting records stored in leaf nodes. Our subsequent analysis will show that existing decision tree construction algorithms just consider the cost of visiting intermediate nodes. This is intuitive because a decision tree construction algorithm will not distinguish two leaf nodes with different number of records if each node contains only records of the same class. Interestingly, our analysis will also show that the existing categorization algorithm proposed in [7] considers the cost of visiting leaf nodes, but not the cost of visiting intermediate nodes generated by future splits. This paper proposes an algorithm that considers both costs.

We now describe how we construct a navigational tree. Since the problem of finding a tree with minimal navigational cost is NP hard, we propose an approximate algorithm in Figure 8.

The algorithm is similar to decision tree construction algorithms [21]. The algorithm only considers attributes that appear in search conditions of some queries in H as splitting attributes. The intuition is that if no user is using an attribute then that attribute is not interesting to any user.

The algorithm starts from the root of the tree. Each record has a class label assigned in the clustering step. If all records in the tree have the same class label, the algorithm stops (line 2). Otherwise, it selects the attribute that gives the maximal gain (Section 5.2 will discuss how to compute the gain) to expand the tree (line 3 to 12).

For a categorical attribute, a new subtree will be created with one branch for each value of the attribute, and the gain will be computed over that subtee (line 4 to 6). In case a categorical attribute may have too many values and thus generate too many branches, we can add intermediate levels to that attribute (e.g., location can be divided into region, state, county, zip code levels).

For a numeric attribute A_i, the split can be binary (i.e., $A_i <= v$ or $A_i > v$), or multi-way (i.e., use ranges $v_j <= A_i < v_{j+1}$, $j = 1, \ldots$). The binary split only considers all possible locations and selects the one that generates the best partition based on the criteria. Multi-way split, however, needs to consider all possible ranges of the attribute values of A, which can be very large. Thus we use binary split in this paper. For a numerical value attribute, the algorithm will generate one subtree for every possible split point, and compute a gain for that split point (line 7 to 10). The best split point will be selected and the gain of the best split is the gain of the attribute. Once the attribute with the maximal gain is selected, if the gain exceeds a user-defined threshold ϵ, the tree will be expanded by adding the selected subtree to the current root (line 13-14). The algorithm continues to expand every leaf of the new subtree at line 14. Since a categorical attribute can only generate

BuildTree(A_H,D_Q,C,ϵ) where A_H is the set of attributes appeared in search conditions of some queries in H, D_Q is query results, C is the class labels assigned in the clustering step for each record in D_Q, and ϵ is a user defined stopping threshold.
1. create a root r.
2. if all records in D_Q have the same class label, stop.
3. for each attribute $A_i \in A_H$.
4.   if A_i is a categorical attribute then
5.   for each value v of A_i, create a branch under the current root. Add those records with $A_i = v$ to that branch.
6.   compute split gain $g(A_i, T_i)$ where T_i is the subtree created.
7.   else
8.   for each value v of A_i, create a tree T_i^v with r as the root and two branches, one for those records with $A_i \le v$, one for those records with $A_i > v$.
9.   compute split gain $g(A_i, T_i^v)$ for each split point v and choose the maximal one as $g(A_i, T_i)$.
10.   end if
11. end for
12. choose the attribute A_j with the maximal $g(A_j, T_j)$. Remove A_j from A_H if A_j is categorical.
13. If $g(A_j, T_j) > \epsilon$ then
14.   Replace r with the subtree T_j. for each leaf n_k in T_j with records in D_{Qk}, BuildTree(A_H,D_{Qk},C,ϵ).
15. end if

Figure 8: Algorithm 2
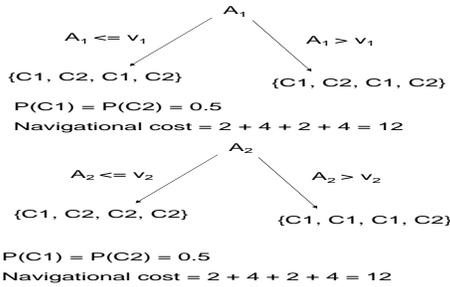
(a) An example for the need of considering future splits.
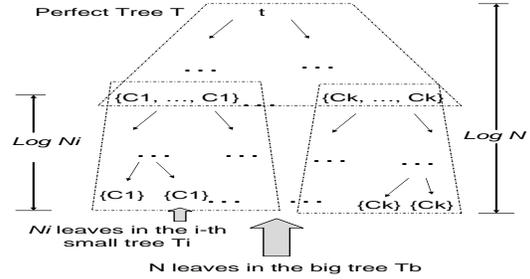
(b) The explanation for Observation 1.

Figure 9: Cost of visiting intermediate nodes.

one possible split, it will be removed from A_H if it is selected as the split attribute.

5.2 Splitting Criteria

The main difference between Algorithm 2 and the existing decision tree construction algorithm is how to compute the gain of a split. Existing decision tree construction algorithms such as C4.5 [21] compute an information gain to measure how good an attribute classifies data. However, this paper wants to reduce the navigational cost, which consists of the cost of visiting intermediate nodes and the cost of visiting records in the leaves. Our analysis will show that information gain ignores the cost of visiting records, and the existing navigational tree construction algorithm [7] ignores the cost of visiting intermediate nodes generated by future splits.
Cost of visiting leaves: Let t be the node to be split and $N(t)$ be the number of records in t. Let t_1 and t_2 be children generated by a split. Let P_i be the probability that users are interested in cluster C_i. The gain equals the reduction of navigational cost when t is split into t_1 and t_2. Thus based on the navigational cost defined in Definition 2, the reduction of the cost of visiting records due to splitting t into t_1 and t_2 equals

$$N(t) \sum_{C_l \cap t \neq \emptyset} P_l - \sum_{j=1,2} N(t_j)(\sum_{C_i \cap t_j \neq \emptyset} P_i) \quad (1)$$

Existing decision tree construction algorithms do not consider the cost of visiting leaf records. For example, consider a split that generates two nodes that contain records with labels (C_1, C_2, C_1) and (C_2), and a split that generates two nodes that contain records with labels (C_2, C_1, C_2) and (C_1). These two splits have the same

information gain (will be described later). However, if $P_1 = 0.5$, and $P_2 = 0$, then the navigational cost for the first split is smaller because the cost is 1.5 for the first split and is 2 for the second split.
Cost of visiting intermediate nodes: Now we show how to estimate the reduction of the cost of visiting intermediate nodes. The existing work [7] directly computes this cost reduction by treating the nodes generated by the split (i.e., t_1 and t_2) as leaves. However, this is problematic because t_1 and t_2 may get split further.

For example, Figure 9 (a) shows a node with 8 records, of which 4 belong to C_1, and 4 belong to C_2. Let us assume that the first split using attribute A_1 generates two nodes each of which contains records with class labels: (C_1, C_2, C_1, C_2), and the second split using a different attribute A_2 generates two nodes containing records with class labels: (C_1, C_1, C_1, C_2) and (C_2, C_2, C_2, C_1). As illustrated in Figure 9 (a), the navigational cost is the same for both splits if the new nodes are considered leaves. However, the second split is better because its leaf nodes have a lower degree of impurity than leaf nodes generated by the first split, and is likely to have a lower navigational cost after future splits.

Next we examine how to estimate the cost of visiting intermediate nodes. Since it is infeasible to consider all possible trees rooted at t, we only consider the set of trees that perfectly classify records in t, i.e., their leaves only contain records of one class. We call these trees *perfect* trees. Obviously, perfect trees can not be split further. We have the following observation.

OBSERVATION 1. *Given a perfect tree T with N records and k classes, where each class C_i in T has N_i records. The entropy $E(t) = -\sum_{1 \le i \le k} \frac{N_i}{N} \log \frac{N_i}{N}$ approximates the average length of root-to-leaf paths for all records in T.*

This observation can be explained as illustrated in Figure 9 (b). Since T is a perfect tree, its leaves contain only one class per node. For each such leaf L_i that contains N_i records of class C_i, we can further expand it into a smaller subtree T_i which is rooted at L_i, and its leaf contains exactly one record in C_i. Each such small subtree T_i contains N_i leaves. All these subtrees and T compose a big tree T_b that contains $\sum_{1 \le i \le k} N_i = N$ leaves. We further assume that each T_i and the big tree T_b are balanced, thus the height for T_i is $logN_i$ and the height for T_b is $logN$. Note that for the i-th leaf L_i in T, the length of the path from root to L_i equals the height of big tree T_b minus the height of small tree T_i. There are N_i records in L_i, all with the same path from the root. Thus the average length of root-to-leaf paths for records in T equals

$$\sum_{1 \le i \le k} \frac{N_i}{N}(logN - logN_i) = -\sum_{1 \le i \le k} \frac{N_i}{N} log \frac{N_i}{N}. \qquad (2)$$

This is exactly the entropy $E(t)$. Note that most existing decision tree algorithms choose the split that maximizes information gain. Information gain is the reduction of entropy due to a split and is represented in the following formula.

$$IGain(t, t_1, t_2) = E(t) - \frac{N_1}{N}E(t_1) - \frac{N_2}{N}E(t_2) \qquad (3)$$

Thus a split with a high information gain will generate a tree with a low entropy. Based on Observation 1, this tree will have short root-to-leaf paths as well. Thus Observation 1 agrees with the fact that the decision tree construction algorithms prefer shorter trees over longer trees [19]. Since the cost of visiting intermediate nodes equals the product of path lengths and fan-out in Definition 2, if we assume the average fan-out is about the same for all trees, then the cost of visiting intermediate nodes is proportional to the length of root-to-leaf paths. Therefore, we can use information gain to estimate the cost reduction of visiting intermediate nodes.

Combining both costs: The remaining problem is how to combine the two types of costs. One can certainly assign certain weights to both costs. Here we take a normalization approach, which uses the following formula to estimate the gain of splitting t into t_1 and t_2.

$$\frac{IGain(t, t_1, t_2)/E(t)}{(\sum_{j=1,2} N(t_j)(\sum_{C_i \cap t_j \ne \emptyset} P_i))/(N(t) \sum_{C_l \cap t \ne \emptyset} P_l)} \qquad (4)$$

The denominator is the cost of visiting leaf records after split normalized by the cost before split. A split always reduces the cost of visiting records (the proof is straightforward and is omitted due to lack of space). Thus the denominator ranges from (0,1]. The nominator is the information gain normalized by the entropy of t. We compute a ratio between these two terms rather than a sum of the nominator and (1-denominator) because in practice the nominator (information gain) is often quite small. Thus the ratio is more sensitive to the nominator when the denominator is similar. For example, suppose the nominators for two splits are 0.1 and 0.2, and the denominator is 0.5 in both cases. The sum equals 0.6 and 0.7, respectively. The difference is 0.1. The ratio equals 0.2 and 0.4, respectively, and the difference is 0.2.

Complexity analysis: We implemented Algorithm 2 by modifying the well-known C4.5 decision tree construction algorithm [21]. Let n be the number of records in query results, m be the number of attributes, and k be the number of classes. The gain in Formula 4 can be computed in $O(k)$ time. C4.5 also uses several optimizations such as computing the gains for all split points of an attribute in one pass, sorting all records on different attribute values beforehand, and reusing the sort order. The cost of sorting records on different attribute values is $O(mn \log n)$, and the cost of computing gains for all possible splits at one node is $O(mnk)$ because there

are at most m split attributes and n possible split points, and each gain can be computed in $O(k)$ time. If we assume the generated tree has $O(\log n)$ levels, the total time is $O(mnk \log n)$. Note that C4.5 is an in-memory algorithm, which is sufficient when the results of queries can fit in memory. There exist efficient disk-based algorithms [14], and our implementation can be easily extended to using such algorithms.

Finally we illustrate how the tree for Example 1 (in Figure 2) is generated. Suppose the first step of our approach creates 3 clusters: C_1 for high return funds, C_2 for low risk funds, and C_3 for the remaining funds. Let us assume $P_1 = P_2 = 0.5$, and $P_3 = 0$. The tree construction algorithm selects "manager year > 9.9" as the first split because it generates the highest gain. This is intuitive because records in both C_1 and C_2 (high returns or low risks funds) belong to the right branch and records in C_3 belong to the left branch. It is easy to verify that both the cost of visiting records and the cost of visiting intermediate nodes get reduced. The right branch is split further using "1 Yr return > 19.27", where the records in C_1 and C_2 are separated. The algorithm then stops.

6. EXPERIMENTAL EVALUATION

In this section we present the results of an empirical study to evaluate our approach against existing approaches.

6.1 Experimental Setup

We conducted our experiments on a machine with Intel PIII 3.2 GHZ CPU, 2 GB RAM, and running Windows XP.

Dataset: We used a data set of 18537 mutual funds downloaded from *www.scottrade.com*. The dataset includes almost *all* available mutual funds in US market. There are 33 attributes, 28 numerical and 5 categorical. The total data size is 20 MB.

Query history: We collected query history by asking 14 users (students in our department) to ask queries against the dataset. Each query had 0 to 6 range conditions. The history contained 124 queries with uniform weight. Each user asked about the same number of queries. There were six attributes "1 year return", "3 year return", "standard deviation", "minimal investment", "expenses", and "manager year" in search conditions of these queries. Each query had 4.4 conditions on average. We assume each query has equal weight. We did observe that users started with a general query which returned many answers, and then gradually refined the query until it returned a small number of answers.

Algorithms: We implemented Algorithm 1 in C++. The clusters are stored by adding a column to the data table to store the class labels of each record. The distance threshold B in the merging algorithm is set to 0.2. We implemented the tree construction algorithm (Algorithm 2) by modifying C4.5 [21] Release 8 (available at http://www.rulequest.com/Personal/). The stopping parameter ϵ in Algorithm 2 is set to 0.001. We have developed a tool that allows users to navigate query results using generated trees. Users can also sort the results in a leaf node by any attribute.

We compare Algorithm 1 with the algorithm proposed in [7], referred to as No-Clustering Algorithm. It differs from our algorithm on two aspects: (1) it does not consider different user preferences, (2) it does not consider the cost of visiting intermediate nodes generated by future splits.

We also compare against an algorithm that first uses the clustering step to generate class labels, then uses standard C4.5 to create the navigational tree. This algorithm is called Clustering+C4.5.

Setup of user study: We conducted an empirical study by asking 47 subjects (with no overlap with the 14 users giving the query history) to use this tool. The subjects were randomly selected students. Each subject was first given a tutorial about mutual funds and how

Table 2: Test queries used in user study

Query Condition	Result size
Q1: Fund Name like '%Vanguard%'	193
Q2: Manager Year $\geq$ 20	140
Q3: Standard Deviation $\leq$ 0.05	352
Q4: All funds	18537

Table 3: Results of Survey

Algorithm	# subjects that called it best
Algorithm 1	37
Clustering+C4.5	6
No-Clustering	0
Did not respond	4

to use this tool. Next, each subject was given the results of 4 test queries listed in Table 2, which do not appear in the query history. For each such query, the subject was asked to navigate the trees generated by the three algorithms mentioned above, and to select 1-20 funds that he would like to invest. The trees were also presented to the subject in a random order to reduce the possible bias introduced by the order of trees. Finally each subject was asked to rank the three algorithms on its effectiveness.

Metrics: We use three metrics to measure the quality of categorization methods. The first metric is the total navigational cost defined below. Unlike the probabilistic navigational cost in Definition 2, this cost is the real count of intermediate nodes (including siblings) and records visited by a subject. We use equal weight for visiting intermediate nodes and visiting records by setting $w_1 = w_2 = 1$
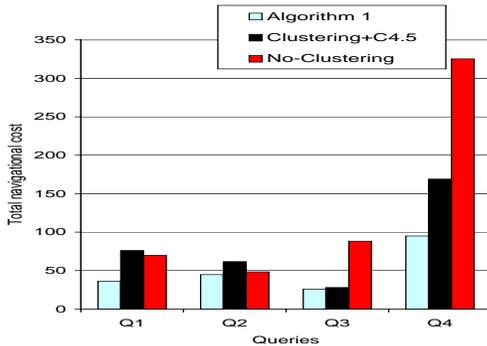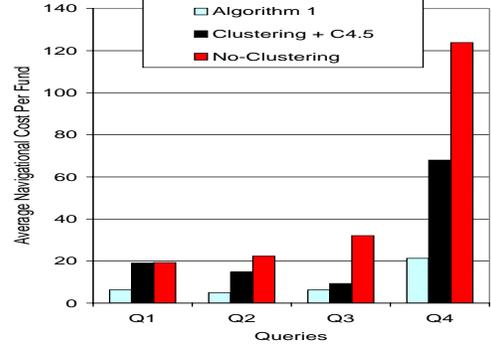
$$\sum_{\forall \text{ leaf t visited by a subject}} (w_1 N(t) + w_2 \sum_{t_i \in Anc(t)} |Sib(t_i)|). \quad (5)$$

In general the lower the total navigational cost, the better the categorization method. The second metric is the number of funds selected as worth investing by a subject. In general a good categorization method shall make it easy for a subject to navigate the results, leading to a higher number of funds selected. The third metric is the average navigational cost per selected fund. This normalizes the total navigational cost against the number of funds selected.

6.2 Overall Comparison with Existing Categorization Methods

Results of user study: Figure 10 reports the total navigational cost, averaged over all the subjects, for Algorithm 1, Clustering+C4.5, and No-clustering. Figure 11 reports the average navigational cost per selected fund for these algorithms. Figure 12 reports the average number of funds selected by each subject (i.e., considered worth investing by the subject).

The results show that the navigational trees generated by Algo-

Figure 11: Average navigational cost
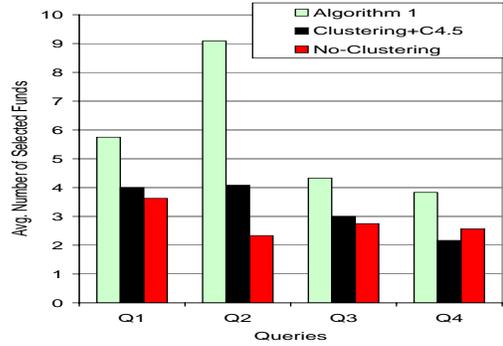
Figure 12: Average number of selected funds per subject

rithm 1 have the lowest total navigational cost and the lowest average cost per selected fund. Users have also found more funds worth investing using our algorithm than the other two algorithms, suggesting our method makes it easier for users to find interesting funds. The improvements of Algorithm 1 in terms of total navigational cost is lower than the improvements in terms of average navigation cost per fund, because using Algorithm 1 users have selected more funds to invest than the other two methods.

The trees generated by No-clustering have the worst results. This is expected because No-Clustering ignores different user preferences, and does not consider future splits when generating navigational trees. Clustering+C4.5 also have worse results than our method. The reason is that our algorithm uses a splitting criterion that considers the cost of visiting records, while standard decision tree algorithms do not. The improvement is less significant for query Q3 because the results of Q3 contain very few classes, thus the differences between these trees are smaller than the differences between trees for other queries.

We also ran a paired t-test over the average navigational costs

Figure 10: Total navigational cost
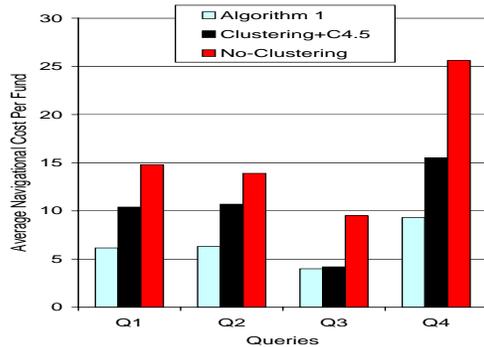
Figure 13: Average navigational cost for funds recommended by experts
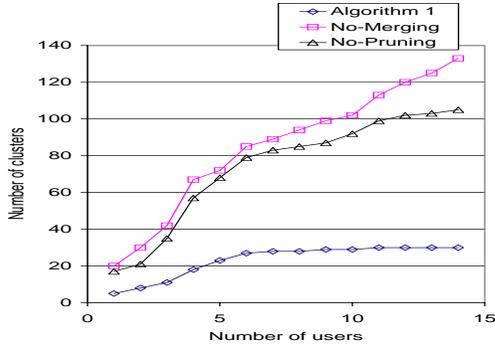
Figure 14: Number of clusters generated when varying number of users

Figure 15: Impact of pruning and merging on average navigational cost

of all subjects and all test queries for each method to verify the difference is significant. The probability of the null hypothesis that Algorithm 1 and Clustering+C4.5 have the same average cost is about 0.07, and probability of that Algorithm 1 and No-clustering have the same average cost is about 0.04. Thus the improvement of Algorithm1 over the other two methods is statistically significant.

The results show that using our approach, on average a user only needs to visit no more than 10 records or intermediate nodes for queries Q1, Q2, and Q3 to locate an interesting record, and needs to visit about 22 records or intermediate nodes for query Q4 that returns all records. The total navigational cost for our algorithm is less than 50 for Q1, Q2, and Q3, and is less than 100 for Q4 (about 0.5% of the result size). This is clearly better than browsing hundreds of results for the first 3 queries and tens of thousands of results for the last query.

Table 3 reports the number of subjects that called our algorithm the best among the three algorithms. The results show that a majority of subjects considered our algorithm the best.

A sanity check: We also conducted a sanity check. The goal is to examine how our approach works when the selected funds are given by domain experts. Some investment research companies such as MorningStar and ValueLine recommend "top performing" mutual funds. We obtained the list of 1086 top performing mutual funds from ValueLine and assumed that there is a fictitious user that will select these funds with 100% probability. We computed the cost of visiting the intersection of these top performing funds with the results of the four test queries in Table 2 using Formula 5. Figure 13 shows the average navigational costs per selected fund for the three algorithms. We do not report total navigational cost because the number of funds selected is the same for all methods. The results are similar to the results of the user study: our approach leads to the lowest navigational cost. Note that the focus of our approach is not to select the "best" funds for users like those companies giving recommendations. Our focus is to find the best navigational trees to help users quickly distinguish different types of funds and select those meeting their needs. Users have the final say in our methods. Further, a user may not agree with the way in which those companies select a "top performing" funds, and those companies do not recommend relatively new funds. Finally, such expert recommendations may be unavailable for many data sets where our method can be applied.

6.3   Evaluation of The Clustering Step

This section evaluates the clustering step in our approach. The clustering step utilizes a number of optimization techniques to deal with large data sets and large query histories. The techniques include query pruning and query merging, where the query merging
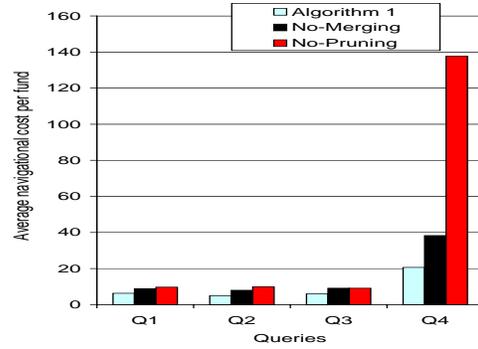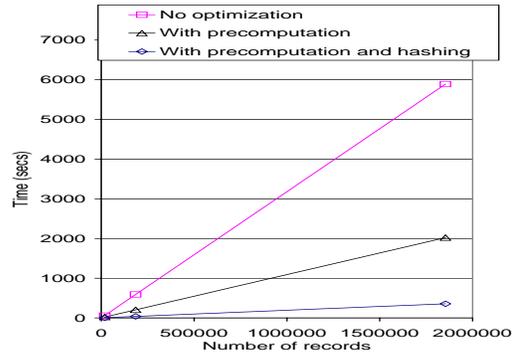

Figure 16: Merging time when varying number of records

also uses precomputation and min-wise hashing techniques. This section evaluates the effectiveness of these techniques.

Query pruning and merging: We first study the impact of query pruning and merging on the number of clusters generated. We choose a random order of users who provide the query history, and then vary the query history by including different number of users. We have also tried different orders but the results are similar. Figure 14 reports the number of clusters generated using our method. We also turn off query pruning and query merging separately and report the results. There are two interesting observations. First, the results show that the number of clusters drops dramatically by using pruning and merging techniques. Second, as the number of users increases, the total number of clusters generated by our method increases less quickly and seems to converge to around 30 clusters. We examine the clusters generated and find that the queries asked by later users tend to be very similar to the queries asked by previous users, and can be merged with these previous queries. This shows that user preferences (represented in the format of clusters)
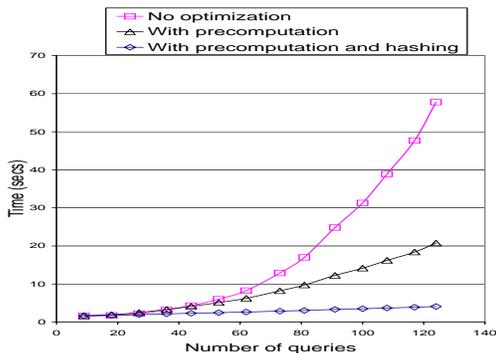

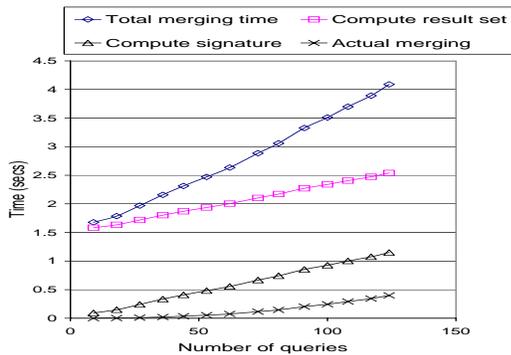**Figure 17: Merging time when varying query history sizes**

Figure 18: Breakdown of merging time

tend to converge as the number of users increases. Thus it is possible to summarize the preferences of a large number of users.

Next we investigate the impact of the pruning and merging techniques on the quality of trees constructed using generated clusters. Rather than asking the subjects to redo the user study, we simulated the user study as follows. We first recorded the set of funds considered worth investing for each subject. We assume that each subject will select these funds with 100% probability. We then use Formula 5 to compute the navigational cost to visit these funds using the trees created from the clusters generated without pruning. We simulated the user study in the same way as above for the trees generated without query merging. Figure 15 reports the average navigational cost of these trees. Note that in the simulation the number of funds selected by a subject for each method is the same, thus we do not report the total navigational cost and the number of funds selected.

The results show that both query pruning and query merging reduced the navigational cost, especially when the result size is large (Q4). Without pruning or merging, too many clusters are generated, and many records in the results of very similar queries are placed in different clusters. Thus the navigational trees using these clusters are usually much bigger than the trees generated using fewer clusters, leading to higher navigational cost. Further, without pruning, all queries including those returning many answers are considered as the evidence of user interests. Thus the probabilities that user are interested in those clusters are often overestimated, leading to higher navigational costs because users will spend more time on those less interesting clusters. In both cases, this problem is more severe when the result size is large (Q4) because the query results contain more clusters.

Precomputation and min-wise hashing: Query merging is the most expensive part in the clustering step (generating clusters and query pruning took no more than 2 seconds in our experiments). Next we study the impact of precomputation and min-wise hashing on merging time. We first examine the merging time when increasing the number of records. We have generated two synthetic data sets by duplicating records in the original data set 10 and 100 times. In this way, the synthetic data sets have the same clusters as the original data sets. Figure 16 reports the execution time of our merging algorithm that uses both precomputation and min-wise hashing, along with the execution time of the merging algorithm without both optimizations, and the merging algorithm with precomputation but without min-wise hashing. The results show that all three algorithms scale linearly to the number of records, which is expected. Using precomputation and min-wise hashing does greatly reduce the execution time. For example, our algorithm took about 6 minutes for the data set with 1853700 rows while the algorithm without optimization took 98 minutes.

We next examine the merging time when varying the query his-

tory size. As mentioned before, we have picked a random order of users and generated smaller query histories by including queries asked by the first few users. Figure 17 reports the merging time for the three algorithms above. For very small query histories (no more than 40 queries), all algorithms took about the same time. However, as query history size increases beyond 40, both precomputation and min-wise hashing greatly reduce the merging time. For example, our algorithm took about 4.1 seconds for the query history with 124 queries while the algorithm without optimization took about 58 seconds.

Figure 18 reports the breakdown of the time for individual steps in our algorithm. The results show that the time to compute result sets and the time to generate signatures increase linearly with the query history size. The time for actual merging operations does increase faster (in the order of $|H'|^2 \log |H'|$), but accounts for a small fraction of the total time for the query histories we consider. In practice if the query history is very large, we may use sampling technique discussed in Section 4 to reduce the query history size and then run our merging algorithm. Further, the clustering step occurs offline and only needs to be done once. We have also examined the clusters generated by all algorithms and there is no difference between them, meaning the min-wise hashing technique is accurate enough.

6.4 Evaluation of The Tree Construction Step

This section evaluates the splitting criterion of our tree construction algorithm as well as the tree construction time. Our algorithm uses a splitting criterion that considers both the cost of visiting intermediate nodes and leaf nodes. Section 6.2 has shown the value of considering leaf nodes as we compare our algorithm with C4.5+Clustering, which does not consider the cost of visiting leaf nodes.

Value of considering the cost of visiting intermediate nodes: Next we evaluated the value of considering the cost of visiting intermediate nodes. We modified the computation of gain in Algorithm 1 to consider only the cost of visiting records in leaves by replacing the nominator in Formula 4 in Section 5 with a constant 1. We simulated the user study in the same way as described in Section 6.3. Figure 19 reports the average navigational cost using the trees generated by this algorithm and the algorithm that considers the cost of visiting intermediate nodes. The results show that considering the cost of visiting intermediate nodes does reduce the navigational cost.

Tree Construction Time: Figure 20 and Figure 21 report the tree construction time of our algorithm for the four test queries. Our algorithm took no more than 0.03 second for the first 3 queries that returned several hundred results, and took about 1.4 seconds for the last query that returned 18537 records. Thus our algorithm can be used in an interactive environment.

7. CONCLUSIONS

In this paper, we propose a novel approach to address diverse user preferences when helping users navigate SQL query results. This approach first summarizes preferences of all users in the system by dividing data into clusters using query history. When a specific user asks a query, our approach uses a cost-based algorithm to create a navigational tree over the clusters appearing in the results of the query to help users navigate these results. An empirical study shows the effectiveness of our approach. There are three key differences with existing literature: (1) our approach is the first to address diverse user preferences for categorization approach, (2) our approach does not require a user profile or a meaningful query when deciding the user preferences for a specific user, and (3) the
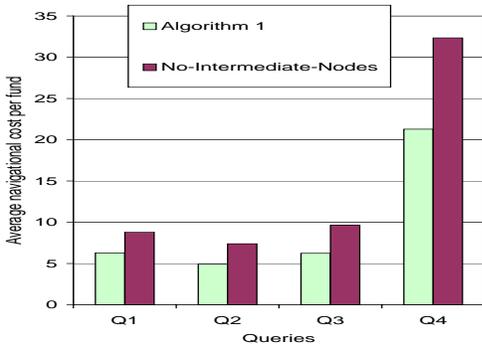
Figure 19: With considering intermediate nodes vs. Without

Figure 20: Execution time for Q1 to Q3

Figure 21: Execution time for Q4

navigational tree construction algorithm considers both the cost of visiting intermediate nodes and leaf nodes. In the future, we plan to further investigate (1) how to use multi-way splits for numerical attributes, (2) how to adapt to the dynamic nature of user preferences (e.g., how to dynamically update the set of clusters stored in the system when user preferences change), (3) how does our approach compare to ranking approach.

8. REFERENCES

[1] R. Agrawal, R. Rantzau, and E. Terzi. Context-sensitive ranking. In *SIGMOD Conference*, pages 383–394, 2006.

[2] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[3] C. Ahlberg and B. Shneiderman. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In *Human Factors in Computing Systems. Conference Proceedings CHI'94*, pages 313–317, 1994.

[4] L. Breiman, J. Friedman, C. J. Stone, and R. Olshen. *Classification and Regression Trees*. CRC Press, 1984.

[5] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[6] S. Card, J. MacKinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.

[7] K. Chakrabarti, S. Chaudhuri, and S. won Hwang. Automatic categorization of query results. In *SIGMOD*, pages 755–766, 2004.

[8] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects based on relationships. In *SIGMOD Conference*, pages 371–382, 2006.

[9] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *VLDB*, pages 888–899, 2004.

[10] G. Das, V. Hristidis, N. Kapoor, and S. Sudarshan. Ordering the attributes of query results. In *SIGMOD*, 2006.

[11] I. S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *KDD*, pages 269–274, 2001.

[12] I. S. Dhillon, S. Mallela, and R. Kumar. Enhanced word clustering for hierarchical text classification. In *SIGKDD*, pages 191–200, 2002.

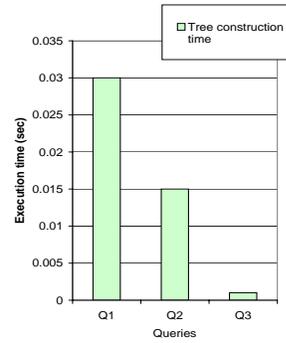[13] L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppin. Placing search in context: the concept revisited. In *WWW '01*, pages 406–414, 2001.
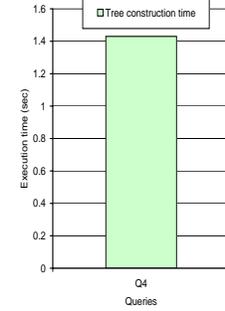
[14] J. Gehrke, V. Ganti, R. Ramakrishnan, and W.-Y. Loh. Boat-optimistic decision tree construction. In *SIGMOD Conference*, pages 169–180, 1999.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[16] T. Joachims. Text categorization with support vector machines: learning with many relevant features. In *ECML-98*, pages 137–142, 1998.

[17] T. Joachims. Optimizing search engines using clickthrough data. In *KDD '02*, pages 133–142, 2002.

[18] D. Koller and M. Sahami. Hierarchically classifying documents using very few words. In *Proceedings of ICML-97*, pages 170–178, 1997.

[19] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[20] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[21] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[22] X. Shen, B. Tan, and C. Zhai. Context-sensitive information retrieval using implicit feedback. In *SIGIR*, pages 43–50, 2005.

[23] K. Sugiyama, K. Hatano, and M. Yoshikawa. Adaptive web search based on user profile constructed without any effort from users. In *WWW*, 2004.

[24] M. Tan. Cost-sensitive learning of classification knowledge and its applications in robotics. *Machine Learning*, 13:7–33, 1993.

[25] K. M. Ting. An instance-weighting method to induce cost-sensitive trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(3):659–665, 2002.

[26] L. Tweedie, R. Spence, D. Williams, and R. S. Bhogal. The attribute explorer. In *Conference on Human Factors in Computing Systems, CHI 1994*, pages 435–436, 1994.

[27] Vivisimo.com. Vivisimo clustering engine. http://vivisimo.com/.

[28] H.-J. Zeng, Q.-C. He, Z. Chen, W.-Y. Ma, and J. Ma. Learning to cluster web search results. In *SIGIR*, pages 210–217, 2004.