Automatic Categorization of Query Results

Kaushik Chakrabarti
Microsoft Research
kaushik@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Seung-won Hwang*
University of Illinois
shwang5@uiuc.edu

ABSTRACT

Exploratory ad-hoc queries could return too many answers – a phenomenon commonly referred to as "information overload". In this paper, we propose to automatically categorize the results of SQL queries to address this problem. We dynamically generate a labeled, hierarchical category structure – users can determine whether a category is relevant or not by examining simply its label; she can then explore just the relevant categories and ignore the remaining ones, thereby reducing information overload. We first develop analytical models to estimate information overload faced by a user for a given exploration. Based on those models, we formulate the categorization problem as a cost optimization problem and develop heuristic algorithms to compute the min-cost categorization.

1. INTRODUCTION

Database systems are being increasingly used for interactive and exploratory data retrieval [1,2,8,14]. In such retrieval, queries often result in too many answers. Not all the retrieved items are relevant to the user; typically, only a tiny fraction of the result set is relevant to her. Unfortunately, she often needs to examine all or most of the retrieved items to find those interesting ones. This too-many-answers phenomenon is commonly referred to as *"information overload"*. For example, consider a real-estate database that maintains information like the location, price, number of bedrooms etc. of each house available for sale. Suppose that a potential buyer is looking for homes in the Seattle/Bellevue Area of Washington, USA in the $200,000 to $300,000 price range. The above query, henceforth referred to as the "Homes" query, returns 6,045 homes when executed on the MSN House&Home home listing database. Information overload makes it hard for the user to separate the interesting items from the uninteresting ones, thereby leading to a huge wastage of user's time and effort. Information overload can happen when the user is not certain of what she is looking for. In such a situation, she would pose a *broad query* in the beginning to avoid exclusion of potentially interesting results. For example, a user shopping for a home is often not sure of the exact neighborhood she wants or the exact price range or the exact square footage at the beginning of the query. Such broad queries may also occur when the user is naïve and refrains from using advanced search features [8]. Finally, information overload is inherent when users are interested in *browsing* through a set of items instead of searching among them.

In the context internet text search, there has been two canonical ways to avoid information overload. First, they group the search results into separate categories. Each category is assigned a descriptive label examining which the user can determine whether the category is relevant or not; she can then click on (i.e., explore) just the relevant categories and ignore the remaining ones. Second, they present the answers to the queries in a ranked order. Thus, *categorization* and *ranking* present two complementary techniques to manage information overload. After browsing the categorization hierarchy and/or examining the ranked results, users often reformulate the query into a more focused narrower query. Therefore, categorization and ranking are indirectly useful even for subsequent reformulation of the queries.

In contrast to the internet text search, categorization and ranking of query results have received much less attention in the database field. Only recently, ranking of query results has received some attention (see Section 2). But, no work has critically examined the use of categorization of query results in a relational database. This is the focus of this paper.

Categorization of database query results presents some unique challenges that are not addressed in the approaches taken by likes of search engines/web directories (Yahoo!, Google) and/or product catalog search (Amazon, Ebay). In all the above cases, the category structures are created *a priori*. The items are tagged (i.e., assigned categories) in advance as well. At search time, the search results are integrated with the pre-defined category structure by simply placing each search result under the category it was assigned during the tagging process. Since such categorization is *independent* of the query, the distribution of items in the categories is *susceptible to skew*: some groups can have a very large number of items and some very few. For example, a search on 'databases' on Amazon.com yields around 34,000 matches out of which 32,580 are under the "books" category. These 32,580 items are not categorized any further[1] (can be sorted based on price or publication date or customer rating) and the user is forced to go through the long list to find the relevant items. This defeats the purpose of categorization as it retains the problem of information overload.

In this paper, we propose techniques to *automatically* categorize the *results* of SQL queries on a relational database in order to reduce information overload. Unlike the "a priori" categorization techniques described above, we generate a labeled hierarchical category structure automatically *based on the contents of the tuples in the answer set*. Since our category

SIGMOD 2004, June 13–18, 2004, Paris, France.

*Work done while visiting Microsoft Research.

[1] Typically, the category structure is created *manually* which deters the creation of a detailed category structure. That is why there are no subcategories under books.

structure is generated at query time and hence tailored to the result set of the query at hand, it does not suffer from the problems of a priori categorization discussed above. This paper discusses how such categorization structures can be generated on the fly to best reduce the information overload. We begin by identifying the space of categorizations and develop an understanding of the exploration models that the user may follow in navigating the hierarchies (Section 3). Such understanding helps us compare and contrast the relative goodness of the alternatives for categorization. This leads to an analytical cost model that captures the goodness of a categorization. Such a cost model is driven by the *aggregate knowledge of user behaviors* that can be gleaned from the workload experienced on the system (Section 4). Finally, we show that we can efficiently search the space of categorizations to find a good categorization using the analytical cost models (Section 5). Our solution is general and presents a domain-independent approach to addressing the information overload problem. We perform extensive experiments to evaluate our cost models as well as our categorization algorithm (Section 6).

2. RELATED WORK

OLAP and Data Visualization: Our work on categorization is related to OLAP as both involve presenting a hierarchical, aggregated view of data to the user and allowing her to drill-down/roll-up the categories [13]. However, in OLAP, the user (data analyst) needs to manually specify the grouping attributes and grouping functions (for the computed categories [13]); in our work, those are determined automatically. Information visualization deals with visual ways to present information [6]. It can be thought as a step after categorization to the further reduce information overload: given the category structure proposed in this paper, we can use visualization techniques (using shape, color, size and arrangements) to visually display the tree [6].

Data Mining: Our work on categorization is related to the work on clustering [11,17] and classification [12]. However, there are significant differences between those works and ours. Let us consider clustering first. First, the space in which the clusters are discovered is usually provided there whereas, in categorization, we need to find that space (the categorizing attributes). Second, existing clustering algorithms deal with either exclusively categorical [11] or exclusively numeric spaces [17]; in categorization, the space usually involves both categorical and numeric attributes. Third, the optimization criteria are different; while it is minimizing inter-cluster distance in clustering, it is minimizing cost (information overload) in our case. Our work differs from classification where the categories are already given there along with a training database of labeled tuples and we need predict the label of future, unlabeled tuples [12].

Discretization/Histograms: In the context of numeric attributes, our work is related to the work on discretization [10] and histograms [5,15]. The discretization work assumes that there is a class assigned to each numeric value (as in classification) and uses the entropy minimization heuristic [10]. On the other hand, the histogram bucket selection is based on minimization of errors in result size estimation [5,15].

Ranking: Previous work on overcoming information overload includes ranking and text categorization. Ranked retrieval has traditionally been used in Information Retrieval in the context of keyword searches over text/unstructured data [3] but has been proposed in the context of relational databases recently [2,4,14]. Ranking is a powerful technique for reducing information overload and can be used effectively in complement with categorization. Although categorization has been studied extensively in the text domain [9,16], to the best of our knowledge, this is the first proposal for automatic categorization in the context of relational databases.

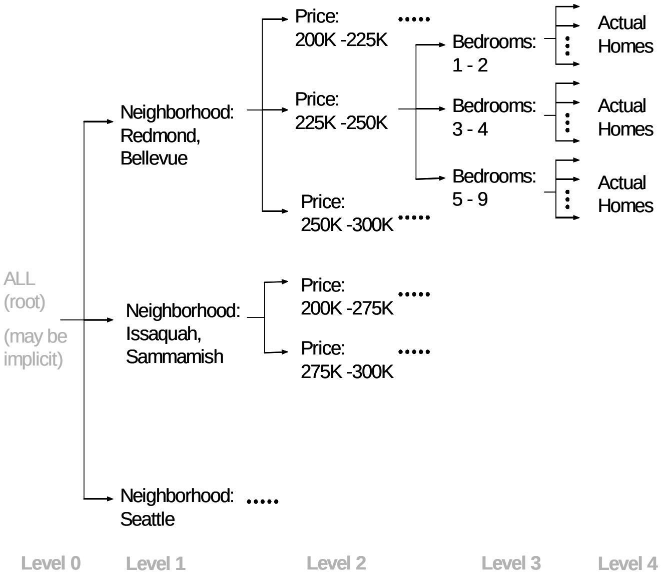


Figure 1: Example of hierarchical categorization of query results (for the "Homes" query in Section 1)

3. BASICS OF CATEGORIZATION

In this section, we define the class of admissible categorizations and describe how a user explores a given category tree.

3.1 Space of Categorizations

Let R be a set of tuples. R can either be a base relation or a materialized view (for browsing applications) or it can be the result of a query Q (for querying applications). We assume that R does not contain any aggregated or derived attributes, i.e., Q does not contain any GROUP BYs or attribute derivations (Q is a SPJ query). A hierarchical categorization of R is a recursive partitioning of the tuples in R based on the data attributes and their values. Figure 1 shows an example of a hierarchical categorization of the results of the "Homes" query presented in Section 1. We define a valid hierarchical categorization T (also referred to as category tree) of R inductively as follows.

Base Case: Given the root or "ALL" node (level 0) which contains all the tuples in R, we partition the tuples in R into an ordered list of mutually disjoint categories (level 1 nodes[2]) using a single attribute. For example, the root node in Figure 1 is partitioned into 3 mutually disjoint categories using the "Neighborhood" attribute: "Neighborhood: Redmond, Bellevue" followed by "Neighborhood: Issaquah, Sammamish" followed by "Neighborhood: Seattle".

Inductive Step: Given a node C at level (*l-1*), we partition the set of tuples *tset(C)* contained in C into an ordered list of mutually disjoint subcategories (level *l* nodes) using a single attribute which is *same* for all nodes at level *(l-1).* We partition a node C

[2] We use the term node and category interchangeably in this paper.

only if C contains more than a certain number of tuples. The attribute used is referred to as the *categorizing attribute* of the level *l* nodes and the *subcategorizing attribute* of the level *(l-1)* nodes. For example, "Price" is the categorizing attribute of *all* nodes at Level 2 (also the subcategorizing attribute of *all* nodes at Level 1). Furthermore, once an attribute is used as a categorizing attribute at any level, it is not repeated at a later level, i.e., there is a 1:1 association between each level of T and the corresponding categorizing attribute. We impose the above constraints to ensure that the categorization is simple, intuitive and easily understandable to the user.

Associated with each node C is a *category label* and a *tuple-set* as defined below:

Category Label: The predicate *label(C)* describing node C. For example, the first child of root (rendered at the top) in Figure 1 has label 'Neighborhood $\in$ {Redmond, Bellevue}' (rendered as 'Neighborhood: Redmond, Bellevue' in Figure 1) while the first child of the above category has label 'Price: 200K–225K'.

Tuple-Set: The set of tuples *tset(C)* (called the *tuple-set* of C) contained in C; either appearing directly under C (if C is a leaf node) or under its subcategories (if C is a non-leaf node). Formally, *tset(C)* is the set of tuples, among the ones contained in the parent of C, which satisfy the predicate *label(C)*. In other words, *tset(C)* is the subset of tuples in R that satisfies the *conjunction* of category labels of all nodes on the path from the root to C. For example, in Figure 1, *tset(C)* for the category with label *'Neighborhood: Redmond, Bellevue'* is the set of all homes in R that are located either in Redmond or in Bellevue while *tset(C)* for its child with label *'Price: 200K–225K'* is the set of all homes in R that are located either in Redmond or in Bellevue *and* priced between 200K and 225K.

The label of a category, therefore, solely and unambiguously describes to the user which tuples, among those in the tuple set of the parent of C, appear under C. Hence, she can determine whether C contains any item that is relevant to her or not by looking just at the label and hence decide whether to explore or ignore C. As discussed above, *label(C)* has the following structure:

If the categorizing attribute A is a categorical attribute: *label(C)* is of the form '$A \in B$' where $B \subset dom_R(A)$ ($dom_R(A)$ denotes the domain of values of attribute A in R). A tuple t satisfies the predicate *label(C)* if $t.A \in B$, otherwise it is false (t.A denotes the value of tuple t on attribute A).

If the categorizing attribute A is a numeric attribute: *label(C)* is of the form '$a_1 \leq A < a_2$' where $a_1, a_2 \in dom_R(A)$. A tuple t satisfies the predicate *label(C)* is true if $a_1 \leq t.A < a_2$, otherwise it is false.

So far, we have described the structure of a hierarchical categorization which defines the class of permissible categorizations. To generate a particular instance of hierarchical categorization, we need to do the following *for each level l*:

- Determine the categorizing attribute A for level *l*
- Given the choice A of categorizing attribute for level *l*, for each category C in level *(l-1)*, determine how to partition the domain of values of A in *tset(C)* into disjoint groups and how do order those groups.

We want to choose the attribute-partitioning combination at each level such that the resulting instance T_{opt} has the *least possible* information overload on the user. For that purpose, we first need a model that captures how a user navigates the result set R using a given category tree T.

3.2 Exploration Model

We present two models capturing two common scenarios in data exploration. One scenario is that the user explores the result set R using the category tree T until she finds *every* tuple $t \in R$ relevant to her, i.e., she does not terminate the exploration after she has found some (but not all) relevant tuples. For example, the user may want to find every home relevant to her in the "Homes" query. In order to ensure that she finds every relevant tuple, she needs to examine every tuple and every category label except the ones that appear under categories she deliberately decides to ignore. Another scenario is that the user is interested in just *one* (or two or a few) tuple(s) in R; so she explores R using T till she finds that one (or few) tuple(s). For example, a user may be satisfied if she finds just one or two homes that are relevant to her. For the purpose of modeling, we assume that, in this scenario, the user is interested in just *one* tuple, i.e., the user explores the result set until she finds the first relevant tuple. We consider these two scenarios because they both occur commonly and they differ in their analytical models; we do not consider additional scenarios as the above two represent the two ends of the spectrum of possible scenarios; other scenarios (e.g., user interested in two/few tuples) fall in between these two ends.

```
Explore C
if C is non-leaf node
    CHOOSE one of the following:
     (1)   Examine all tuples in tset(C) // Option SHOWTUPLES
     (2)   for (i=1; i ≤n; i++) // Option SHOWCAT
              Examine the label of ith subcategory C_i
              CHOOSE one of the following:
               (2.1) Explore C_i
               (2.2) Ignore C_i
else // C is a leaf-node
    Examine all tuples in tset(C) //SHOWTUPLES is only option
```

Figure 2: Model of exploration of node C in 'All' Scenario

3.2.1 Exploration Model for 'All' Scenario

The model of exploration of the subtree rooted at an arbitrary node C is shown in Figure 2. The user starts the exploration by exploring the root node. Given that she has decided to explore the node C, if C is a non-leaf node, she *non-deterministically* (i.e., not known in advance) chooses one of the two options[3]:

Option 'SHOWTUPLES': Browse through the tuples in *tset(C)*. Note that the user needs to examine *all* tuples in *tset(C)* to make sure that she finds every tuple relevant to her.

Option 'SHOWCAT': Examine the labels of *all* the *n* subcategories of C, exploring the ones relevant to her and ignoring the rest. More specifically, she examines the label of each subcategory C_i starting from the first subcategory and *non-deterministically* chooses to either explore it or ignore it. If she chooses to ignore C_i, she simply proceeds and examines the next label (of C_{i+1}). If she chooses to explore C_i, she does so *recursively* based on the same exploration model, i.e., by choosing either 'SHOWTUPLES' or 'SHOWCAT' if it is an internal node or by choosing 'SHOWTUPLES' if it is a leaf

[3] We assume that the user interface displays sufficient information (in addition to the category label) to the user so that she can properly decide between SHOWTUPLES and SHOWCAT.

node. After she finishes the exploration of C_i, she goes ahead and examines the label of the next subcategory of C (of C_{i+1}). When the user reaches the end of the subcategory list, she is done. Note that we assume that the user examines the subcategories in the order it appears under C; it can be from top to bottom (as shown in Figure 1) or from left to right depending on how the tree is rendered by the user interface.

If C is a leaf node, 'SHOWTUPLES' is the only option (option 'SHOWCAT' is not possible since a leaf node has no subcategories).

Example 3.1: Here is an example of an exploration on the tree in Figure 1 in the 'ALL' scenario: explore root using SHOWCAT, examine "Neighborhood:Redmond,Bellevue" and explore it using SHOWCAT, examine "Price:200K-225K" and ignore it, examine "Price:225K-250K" and explore it using SHOWTUPLES, examine *all* tuples under "Price:225K-250K", examine "Price:250K-300K" and ignore it, examine "Neighborhood:Issaquah, Sammamish" and ignore it, examine "Neighborhood:Seattle" and ignore it . Note that examining a node means reading its label while examining a tuple means reading all the fields in the tuple.

```
Explore C
if C is non-leaf node
    CHOOSE one of the following:
     (1)  Examine tuples in  tset(C) from beginning till 1st relevant
          tuple found // Option SHOWTUPLES
     (2)  for (i=1; i ≤n; i++) // Option SHOWCAT
               Examine the label of ith subcategory $C_i$
               CHOOSE one of the following:
                (2.1) Explore $C_i$
                (2.2) Ignore  $C_i$
               if (choice = Explore) break; // examine till 1st relevant
else // C is a leaf-node
        Examine all tuples in tset(C) from beginning till 1st relevant
tuple found  // Option  SHOWTUPLES is the only option
```

Figure 3: Model of exploration of node C in 'One' Scenario

3.2.2 Exploration Model for 'One' Scenario

The model of exploration of an arbitrary node C of the tree T is shown in Figure 3. Once again, the user starts the exploration by exploring the root node. Given that the user has decided to explore a node C, she *non-deterministically* chooses one of the two options:

Option 'SHOWTUPLES': Browse through the tuples in *tset(C)* starting from the first tuple in *tset(C)* till she finds the first relevant tuple. In this paper, we do not assume any particular ordering/ranking when the tuples in *tset(C)* are presented to the user.

Option 'SHOWCAT': Examine the labels of the subcategories of C starting from the first subcategory till the first one she finds interesting. As in the 'ALL' scenario, she examines the label of each subcategory C_i starting from the first one and *non-deterministically* chooses to either explore it or ignore it. If she chooses to ignore C_i, she simply proceeds and examines the next label. If she chooses to explore C_i, she does so *recursively* based on the same exploration model. We assume that when she drills down into C_i, she finds *at least one* relevant tuple in *tset(C_i)*; so, unlike in the 'ALL' scenario, the user does not need to examine the labels of the remaining subcategories of C.

If C is a leaf node, 'SHOWTUPLES' is the only option (browse through the tuples in *tset(C)* starting from the first one till she finds the first relevant tuple).

Example 3.2: Here is an example of an exploration using the tree in Figure 1 in the 'ONE' scenario: explore root using SHOWCAT, examine "Neighborhood:Redmond,Bellevue" and explore it using SHOWCAT, examine "Price:200K-225K" and ignore it, examine "Price:225K-250K" and explore it using SHOWTUPLES, examine tuples under "Price:225K-250K" starting with the first one till she finds the first relevant tuple.

4. COST ESTIMATION

Since we want to generate the tree imposes the *least possible* information overload on the user, we need to estimate the information overload that a user will face during an exploration using a given category T. We describe how to estimate that in this section.

4.1 Cost Models

4.1.1 Cost Model for 'ALL' Scenario

Let us first consider the 'ALL' scenario. Given a user exploration X using category tree T, we define *information overload cost*, or simply *cost* (denoted by $\text{Cost}_{\text{All}}(X,T)$), as the total number of items (which includes both category labels and data tuples) examined by the user during X. The above definition is based on the assumption that the time spent in finding the relevant tuples is proportional to the number of items the user needs to examine: more the number of items she needs to examine, more the time wasted in finding the relevant tuples, higher the information overload.

Example 4.1: We compute the cost $\text{Cost}_{\text{All}}(X,T)$ of the exploration in Example 3.1. Assuming 0 cost for examining the root node (for simplicity) and assuming that there are 20 tuples under "Price:225K-250K", the cost is 3 (for examining the labels of the 3 first-level categories) + 3 (for examining the labels of the 3 subcategories of "Neighborhood:Redmond,Bellevue") + 20 (examining the tuples under "Price:225K-250K") = 26.

If we knew the mind of the user, i.e., we *deterministically* knew what choices in Figure 2 a particular user will make (which categories she will explore and which ones she will ignore, when she will use SHOWTUPLES and when SHOWCAT, etc.), we could generate the tree that would minimize the number of items this particular user needs to examine. Since we do not have that user-specific knowledge[4], we use the aggregate knowledge of previous user behavior in order to estimate the information overload cost $\text{Cost}_{\text{All}}(T)$ that a user will face, *on average*, during an exploration using a given category tree T. Based on the definition of information overload, $\text{Cost}_{\text{All}}(T)$ is the number of items (which includes category labels and data tuples) that a user will need to examine, on average, during the exploration of R using T till she finds *all* tuples relevant to her. Subsequently, we

[4] We can get some of this knowledge by observing past behavior of this particular user (known as 'personalization'). We do not pursue that direction in this paper. As a result, our technique does not produce the optimal tree for any user in particular but for the (hypothetical) average user (the tree produced is the same for any user for a given query). Since we are optimizing for the average case, we expect it to be reasonably good, on average, for individual users assuming that the individual users conform to the previous behavior captured by the workload.

can find the category tree that minimizes this average cost of exploration. Since the user choices in Figure 2 are non-deterministic and not equally likely, we need to know the following two probabilities associated with each category of T in order to compute $Cost_{All}(T)$:

Exploration Probability: The probability P(C) that the user exploring T explores category C, using either SHOWTUPLES or SHOWCAT, upon examining its label. The probability that the user ignores C upon examining its label is therefore (1-P(C)).

SHOWTUPLES Probability: The probability $P_w(C)$[5] that the user goes for option 'SHOWTUPLES' for category C *given that she explores C*. The SHOWCAT probability of C, i.e., the probability that the user goes for option 'SHOWCAT' *given that she explores C* is therefore (1- $P_w(C)$). If C is a leaf category, $P_w(C)$ = 1 because given that the user explores C, 'SHOWTUPLES' is the only option.

How we compute these probabilities using past user behavior is discussed in Section 4.2. We next discuss how to compute $Cost_{All}(T)$ assuming we know the above probabilities.

Let us consider a non-leaf node C of T. Let $C_1, C_2, \ldots, C_n$ be the *n* subcategories of C. Let us consider the cost $Cost_{All}(T_C)$ of exploring the subtree T_C rooted at C *given that the user has chosen to explore C*. Since the cost is always computed in the context of a given tree T, for simplicity of notation, we henceforth denote $Cost_{All}(T_C)$ by $Cost_{All}(C)$; $Cost_{All}(T)$ is simply $Cost_{All}(root)$. If the user goes for option 'SHOWTUPLES' for C, she examines *all* the tuples in *tset(C)*, so the cost is |*tset(C)*|. If she goes for option 'SHOWCAT', the total cost is the cost of examining the labels of *all* the subcategories *plus* the cost of exploring the subcategories she chooses to explore upon examining the labels. The first component is K*n where K is the cost of examining a category label relative to the cost of examining a data tuple; the second cost is $Cost_{All}(C_i)$ if she chooses to explore C_i, 0 if she chooses to ignore it. Putting it all together,

$Cost_{All}(C)$ =

$$P_w(C)*|tset(C)| + (1-P_w(C)) * (K*n + \sum_{i=1}^{n} P(C_i)*Cost_{All}(C_i)) \quad (1)$$

If C is a leaf node, $Cost_{All}(C)$ = |*tset(C)*|. Note that the above definition still holds as $P_w(C)$ = 1 for a leaf node.

4.1.2 Cost Model for 'ONE' Scenario

In this scenario, the information overload cost $Cost_{One}(T)$ that a user will face, *on average*, during an exploration using a given category tree T is the number of items that a user will need to examine, on average, till she finds the *first* tuple relevant to her. Let us consider the cost $Cost_{One}(C)$ of exploring the subtree rooted at C *given that the user has chosen to explore C*; $Cost_{One}(T)$ is simply $Cost_{One}(root)$. If the user goes for option 'SHOWTUPLES' for C and *frac(C)* denotes the fraction of tuples in *tset(C)* that she needs to examine, on average, before she finds the first relevant tuple, the cost, on average, is *frac(C)**|*tset(C)*|. If she goes for option 'SHOWCAT', the total cost is (K*i + $Cost_{One}(C_i)$) if C_i is the first subcategory of C explored by the user (since the user examines only *i* labels and explores only C_i). Putting it all together,

$$Cost_{One}(C) = P_w(C)*frac(C)*|tset(C)| + (1-P_w(C)) * \sum_{i=1}^{n} \Big(\text{Prob. that } C_i \text{ is the first category explored}* (K*i + Cost_{One}(C_i))\Big)$$

The probability that C_i is the first category explored (i.e., probability that the user explores C_i but none of C_1 to $C_{(i-1)}$), is $\prod_{j=1}^{(i-1)}$ (1-P(C_j)) *P(C_i), so

$$Cost_{One}(C) = P_w(C)*frac(C)*|tset(C)|$$
$$+ (1-P_w(C)) * \sum_{i=1}^{n} \Big(\prod_{j=1}^{(i-1)} (1-P(C_j)) * P(C_i)* (K*i + Cost_{One}(C_i))\Big) (2)$$

If C is a leaf node, $Cost_{One}(T_C)$ = *frac(C)**|*tset(C)*|; so the above definition still holds as $P_w(C)$ = 1 for a leaf node.

4.2 Using Workload to Estimate Probabilities

As stated in Section 4.1, we need to know the probabilities $P_w(C)$ and P(C) to be able to compute the average exploration cost $Cost_{All}(T)$ (or $Cost_{One}(T)$) of a given tree T (all other variables in the equations (1) and (2) are either constants or known for a given T). To estimate these probabilities automatically (without any input from domain expert), we use the *aggregate* knowledge of previous user behavior. Specifically, we look at the log of queries that users of this particular application have asked in the past (referred to as *'workload'*). Our technique only requires *the log of SQL query strings* as input; this is easy to obtain since the profiling tools that exist on commercial DBMSs log the queries that executes on the system anyway. Since we use the aggregate knowledge, our categorization is the same for all users for the same result set; it only varies with the result set.

Computing SHOWTUPLES Probability: Given that the user explores a non-leaf node C, she has two mutually exclusive choices: do SHOWTUPLES or do SHOWCAT. Let us first consider the SHOWCAT probability of C, i.e., the probability that the user does SHOWCAT given that she explores C. We presume that the user does SHOWCAT for C (given that she explores C) if the *subcategorizing attribute SA(C)* of C is such that user is interested in only a few of the subcategories (i.e., in only a *few values* of *SA(C)*); in this situation, using SHOWCAT enables her to ignore a large fraction of the subcategories and hence significantly cut down the number of tuples she needs to examine. On the other hand, if she is interested in all or most of the subcategories of C, i.e., in *all or most values* of *SA(C)*, she will go for SHOWTUPLES. For example, in Figure 1, suppose the user has decided to explore the "Neigborhood:Redmond,Bellevue" category. If the user is sensitive about 'Price' and cares only about 200K-225K homes, she is most likely to request SHOWCAT and thereby avoid examining the 225K-250K and 250K-300K homes (by ignoring those categories). On the other hand, if she does not care about the 'Price' of the home, the user is most likely to request SHOWTUPLES for that category because she would need to explore all the subcategories if she does SHOWCAT. We estimate the SHOWCAT probability of C using the workload as follows. Suppose that the workload query W_i represents the information need of a user U_i. If U_i has specified a selection condition[6] on *SA(C)* in W_i, it typically means that she is

[5] The subscript 'w' denotes that the user examines the *whole* set of tuples in S_C.

[6] We assume that the workload queries are SPJ queries on a database with star schema, i.e., they are equivalent to select queries on the wide table obtained by joining the fact table with the dimension tables. The attributes and values used in the

interested in a *few* values of *SA(C)*. On the other hand, absence of selection condition on SA(C) means that she is interested in *all* values of SA(C). If $N_{Attr}(A)$ denotes the number of queries in the workload that contain selection condition on attribute *A* and N is the total number of queries in the workload, $N_{Attr}(SA(C))/N$ is the fraction of users that are interested in a few values of *SA(C)*. Assuming that the workload represents the activity of a large number of diverse users and hence forms a good statistical basis for inferencing user behavior, the probability that a random user is interested in a few values of SA(C), i.e., the SHOWCAT probability of C is $N_{Attr}(SA(C))/N$. The SHOWTUPLES probability $P_w(C)$ of C (probability that the user goes for option 'SHOWTUPLES' for C *given that she explores C*) is therefore 1 – $N_{Attr}(SA(C))/N$.

Computing Exploration Probability: We now discuss how we estimate the probability P(C) that the user explores category C, either using SHOWTUPLES or SHOWCAT, upon examining its label. By definition, P(C) = P(User explores C | User examines label of C). Since user explores C implies that user has examined label of C, P(C) = P(User explores C) / P(User examines label of C). Since user examines label iff she explores the parent (say C') of C and chooses SHOWCAT for C',

$$P(C)= \frac{\text{P(User explores C)}}{\text{P(User explores C' and chooses SHOWCAT for C')}}$$

$$= \frac{\text{P(User explores C)}}{\text{P(User explores C')} * \text{P(User chooses SHOWCAT for C'} \mid \text{User explores C')}}$$

P(User chooses SHOWCAT for C'|User explores C') is the SHOWCAT probability of C' = $N_{Attr}(SA(C'))/N$.

A user explores C if she, upon examining the label of C, thinks that there *may* be one or more tuples in *tset(C)* that is of interest to her, i.e., the full path predicate of C (the conjunction of category labels of all nodes on the path from the root to C) is of interest to her. Assuming that the user's interest in a label predicate on one attribute is *independent* of her interest in a label predicate on another attribute, P(User explores C) / P(User explores C') is simply the probability that the user is interested in the label predicate *label(C).*

$$\text{So, } P(C)= \frac{\text{P(User interested in predicate label(C))}}{N_{Attr}(SA(C))/N}$$

Again suppose that the workload query W_i in the workload represents the information need of a user U_i. If W_i has a selection condition on the categorizing attribute CA(C) of C and that selection condition on CA(C) *overlaps* with the predicate *label(C)*, it means that U_i is interested in the predicate *label(C).* If $N_{Overlap}(C)$ denotes the number of queries in the workload whose selection condition on CA(C) *overlaps* with *label(C)*, P(User interested in predicate *label(C)*) = $N_{Overlap}(C)/N$. So, P(C) = $N_{Overlap}(C)/N_{Attr}(SA(C'))$. Since the subcategorizing attribute SA(C') of C' is, by definition, the categorizing attribute CA(C) of C, P(C) = $N_{Overlap}(C)/ N_{Attr}(CA(C))$. We finish this discussion by stating what we mean by *overlap*: if CA(C) (say A) is a categorical attribute, the selection condition "A IN {v_1,…, v_k}" on CA(C) in W_i *overlaps* with the predicate *label(C)* = '$A \in B$' if the two sets {v_1,…, v_k} and B are not mutually disjoint; if CA(C) (again say A) is a numeric attribute, the selection condition "vmin ≤ A ≤ vmax" on CA(C) in W_i *overlaps* with the predicate *label(C)* = '$a_1 \leq A < a_2$' iff the two ranges [vmin,vmax] and [a_1,a_2] overlap.

5. CATEGORIZATION ALGORITHM

Since we know how to compute the information overload cost $Cost_{All}(T)$ of a given tree T, we can enumerate all the permissible category trees on R, compute their costs and pick the tree T_{opt} with the minimum cost. This enumerative algorithm will produce the cost-optimal tree but could be prohibitively expensive as the number of permissible categorizations may be extremely large. In this section, we present our preliminary ideas to reduce the search space of enumeration. We will first present our techniques in the context of 1-level categorization (i.e., a root node pointing to a set of mutually disjoint categories which are not subcategorized any further). In Section 5.2, we generalize that to multi-level categorization.

5.1 One-level Categorization

We now present heuristics to (1) eliminate a subset of relatively unattractive attributes without considering any of their partitionings (Section 5.1.1) and (2) for every attribute selected above, obtain a good partitioning efficiently instead of enumerating all the possible partitionings (Sections 5.1.2 and 5.1.3). Finally, we choose the attribute and its partitioning that has the least cost.

5.1.1 Reducing the Choices of Categorizing Attribute

Since the presence of a selection condition on an attribute in a workload query reflects the user's interest in that attribute (see Section 4.2), attributes that occur infrequently in the workload can be discarded right away while searching for the min-cost tree. Let A be the categorizing attribute chosen for the 1-level categorization. If the occurrence count $N_{Attr}(A)$ of A in the workload is low, the SHOWTUPLES probability $P_w(root)$ of the root node will be high. Since the SHOWTUPLES cost of a tree is typically much higher than its SHOWCAT cost and the choice of partitioning affects only the SHOWCAT cost, a high SHOWTUPLES probability implies that the cost of the resulting tree would have a large first component ($P_w(root)$*|*tset(root)*|) which would contribute to a higher total cost. Therefore, it is reasonable to consider eliminating such low occurring attributes without considering any of their partitionings.

Specifically, we eliminate the uninteresting attributes using the following simple heuristic: if an attribute A occurs in less than a fraction *x* of the queries in the workload, i.e., $N_{Attr}(A)/N < x$, we eliminate A. The threshold *x* will need to be specified by the system designer/domain expert. For example, for the home searching application, if we use *x*=0.4, only 6 attributes, namely neighborhood, price, bedroomcount, bathcount, property-type and square footage, are retained from among 53 attributes in the MSN House&Home dataset. For attribute elimination, we preprocess the workload and maintain, for each potential categorizing attribute A, the number $N_{Attr}(A)$ of queries in the workload that contain selection condition on A. At query time, for each retained attribute, we obtain a good partitioning by invoking the partitioning function discussed in Sections 5.1.2 and 5.1.3 below and choose the attribute-partitioning combination that has the minimum $Cost_{All}(T)$.

5.1.2 Partitioning for Categorical Attributes

We present an algorithm to obtain the optimal partitioning for a given categorizing attribute A that is categorical. Consider the

selection conditions in the queries therefore reflect the user's interest in those attributes and values while searching for items in the fact table.

Attribute A	#queries in workload containing selection condition on A (N_{Attr}(A))
Neighborhood	7327
Price	5210
Bedrooms	6498
SquareFootage	4251
YearBuilt	2347
⋮	⋮

(a) **AttributeUsageCounts table** (one table for the whole database)

Value v_i	# queries in workload whose selection condition on A contains v_i in IN clause ($N_{Overlap}(C_i)$
Seattle,WA	492
Bellevue,WA	1213
Issaquah,WA	547
Redmond,WA	1050
Kirkland,WA	848
⋮	⋮

(b) **OccurrenceCounts table** for Neighborhood attribute (one table for each potential categorizing attribute)

Figure 4: Example of AttributeUsageCounts table and OccurrenceCounts table

case where the user query Q contains a selection condition of the form "A IN {v_1, …, v_k}" on A. We only consider *single-value partitionings* of R in this paper, i.e., we partition R into k categories – one category C_i corresponding to each value v_i in the IN clause (e.g., "Neighborhood:Redmond", "Neighborhood:Bellevue", etc.). The advantage of single-value partitionings is that the category labels are simple and easy to examine; multi-valued categories, on the other hand, would have more complex category labels. Among the single-value partitionings, we want to choose the one with the minimum cost. Since the *set* of categories is identical in all possible single-value partitionings, the only factor that impacts the cost of a single-valued partitioning is the *order* in which the categories are presented to the user. The order affects the cost because our exploration models (see Section 3.2) assume that the user always starts examining the labels of the subcategories from the top and goes downwards (or from left going rightwards depending on the rendering). The cost $Cost_{All}(T)$ to find *all* tuples relevant to her is not affected by the order, so we only consider the cost $Cost_{One}(T)$ to find *one* relevant tuple. It can be shown from Equation (2) that among all possible orderings, $Cost_{One}(T)$ is *minimum* when the categories are presented to the user in *increasing* order of $1/P(C_i) + Cost_{One}(C_i)$; the proof can be found in Appendix A. Intuitively, if the probability $P(C_i)$ of drilling into the category is high, it is better to present it to the user earlier since that would reduce the number of labels of uninteresting categories the user needs to examine. Also, it is better to place the categories with lower exploration cost earlier as that would reduce the overall cost as well. Although the above optimality criterion can be evaluated to obtain the optimal 1-level category tree, it is hard to use the criteria for multilevel category trees due to the complexity of computing $Cost_{One}(C_i)$. However, in contrast, $P(C_i)$ can be evaluated for the multilevel category tree without any complexity. Therefore, we have adopted the heuristic to present the categories in the *decreasing* order of $P(C_i)$, i.e., we only use the first term in the above formula. Although the above is tantamount to assuming equality of $Cost_{One}(C_i)$'s, our experimental results have been encouraging. Recall from Section 4.2 that $P(C_i) = N_{Overlap}(C_i)/N_{Attr}(A)$. Since each category C_i corresponds to a single value v_i, $N_{Overlap}(C_i)$ is the number of queries in the workload whose selection condition on A contains v_i in the IN clause (called the occurrence count $occ(v_i)$ of v_i). To obtain the partitioning, we simply sort the values in the IN clause in the *decreasing* order of $occ(v_i)$.

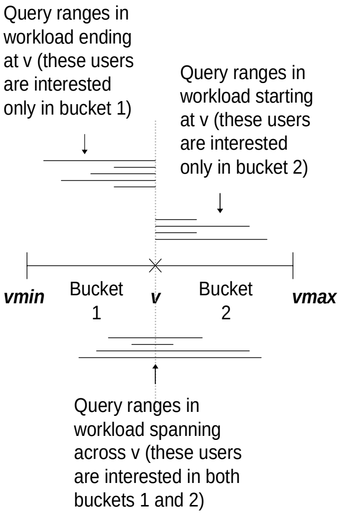
To retrieve the occurrence count $occ(v_i)$ of a given value v_i efficiently, we preprocess the workload and maintain, for each categorical attribute A separately, the occurrence count of each distinct value of A in a database table (see Figure 4(b)). For each occurrence count table, we can build an index on the value to make the retrieval efficient. If necessary, standard compression techniques (e.g., prefix compression, histogramming techniques) can be used to compress the occurrence count table as well.

5.1.3 Partitioning for Numeric Attributes

We present a heuristic to obtain a good partitioning for a given categorizing attribute A that is numeric. Let *vmin* and *vmax* be the minimum and maximum values that the tuples in R can take in attribute A. If the user query Q contains a selection condition on A, *vmin* and *vmax* can be obtained directly from Q. Let us first consider the simple case where we want to partition the above range *(vmin, vmax]* into two mutually disjoint buckets, i.e., identify the best point to split. Let us consider a point *v* (*vmin* < *v* < *vmax*). If a significant number of query ranges (corresponding to the selection condition on A) in the workload *begin* or *end* at *v*, it is a good point to split as the workload suggests that most users would be interested in just one bucket, i.e., either in the bucket *(vmin, v] or* in the bucket *(v, vmax]* but *not both* (see Figure 5(a)). In this situation, she will be able to ignore one of the buckets and hence avoid examining all the items under it, thereby reducing information overload. On the other hand, if few or none of the ranges begin or end at v (i.e., all or most of them *spans* across v), most users would be interested in both buckets (see Figure 5(a)). In this case, the user will end up examining all the items in R and hence incur a high exploration cost; *v* is therefore not a good point to split. If we are to partition the range into *m* buckets, where *m* is specified by the system designer, applying the above intuition, we should select the *(m-1)* points where most query ranges begin and/or end as the splitpoints. However, our cost model suggests that the start and end counts of the splitpoints are not the only factors determining the cost; the other factor is the number of tuples that fall in each bucket. The above splitpoint selection heuristic, therefore, may not always produce the best partitioning in terms of cost, especially if there is a strong correlation between start/end counts of the splitpoints and the number of tuples in each bucket. Such correlations occured rarely in the real-life datasets we used in our experiments and the above heuristic produced low-cost partitionings for those datasets.

Let us consider the point *v* again (*vmin* < *v* < *vmax*). Let $start_v$ and end_v denote the number of query ranges in the workload starting and ending at *v* respectively. We use SUM($start_v$, end_v) as the "goodness score" of the point *v*. The above heuristic is an approximation of the optimal goodness score suggested by the cost model; in the special case where we are selecting a single splitpoint and the resulting two buckets have equal number of tuples, the above heuristic matches the optimal goodness score (i.e., produces the cost-optimal partitioning). Since the above goodness score depends only on *v*, we can precompute the goodness score for each potential splitpoint and store it in a table[7]; Figure 5(b) shows the precomputed goodness scores for all potential splitpoints in the

[7] We assume that the potential splitpoints are separated by a fixed interval (e.g., 1000 in Figure 5(b)).


Query ranges in workload ending at v (these users are interested only in bucket 1)
Query ranges in workload starting at v (these users are interested only in bucket 2)
vmin
Bucket 1
v
Bucket 2
vmax
Query ranges in workload spanning across v (these users are interested in both buckets 1 and 2)

(a)

Splitpoint v	$start_v$	end_v	SUM ($start_v$,end_v)
1000	0	0	0
2000	10	40	50
3000	0	0	0
4000	0	0	0
5000	40	90	130
6000	0	0	0
7000	0	0	0
8000	80	20	100
9000	0	0	0
10000	30	-	-

(b)

Figure 5: (a) Intuition behind splitpoint selection heuristic (b) Part of Splitpoints table for Price attribute (only splitpoints in the range (0, 10000] are shown). The separation interval is 1000.

range (0,10000]). At query time, to produce *m* buckets, we pick the top *(m-1)* splitpoints in the range *(vmin, vmax]* based on goodness scores, leaving out the ones that are unnecessary. Of course, a splitpoint is unnecessary for a range if it contains too few tuples. We will discuss this aspect in greater details in Section 5.2. We examine the splitpoints in the range *(vmin, vmax]* in decreasing order of goodness score; if we get an unnecessary one, we simply skip it (otherwise we select it) and go to the next one till we select *m* splitpoints. Finally, note that the goodness metric may be used as a basis for automatically determining *m* instead of being specified externally.

Example 5.1: In the example in Figure 5(b), if m=3, we will select 5000 and 8000 if both are necessary (since the have the highest goodness values, 130 and 100 respectively). If the splitpoint 8000 is unnecessary, we skip it and pick the next best splitpoint, i.e., 2000; so we will select 5000 and 2000. We always present the categories in ascending order of the values of the split points (i.e., 0-2000 followed by 2000-5000 followed by 5000-10000 in the latter case).

As mentioned before, we preprocess the workload and maintain, for each numeric attribute separately, the goodness scores of each potential splitpoint in a database table (one such *splitpoints* table per numeric attribute as shown in Figure 5(b)); this eliminates the need to access the workload at query time. We index the table based on *v* to speed up the retrieval of goodness scores of the splitpoints in the desired range.

We conclude this section with a note on how the partitioning technique presented above differs from existing histogram techniques. Traditional histogram techniques try to reduce estimation error by grouping together values that has similar source parameter values (e.g., frequencies, areas, spreads) [15]. In contrast, our partitioning tries to reduce cost by grouping together values that are likely to be co-accessed during an exploration, based on workload statistics like access counts.

5.2 Multilevel Categorization

For multilevel categorization, for each level *l*, we need to (1) determine the categorizing attribute A and (2) for each category C in level *(l-1)*, partition the domain of values of A in *tset(C)* such that the information overload is minimized. We partition a node C iff C contains more than M tuples where M is a given parameter. Otherwise, we consider its partitioning unnecessary. There are several advantages to introducing the parameter M: first, it guarantees that no leaf category has more than M tuples (only if there is sufficient number of attributes available for categorization). This is important because, in an interactive environment, we often need to fit all the tuples in a category in the display screen; we can ensure that by choosing M accordingly. We choose M=20 in our user study. Second, it gives the user an opportunity to control the granularity of categorization. A simple level-by-level categorization algorithm is shown in Figure 6.

The algorithm creates the categories level by level (starting with level 0), i.e., all categories at level *(l-1)* are created and added to tree T before any category at level *l*. A new level is necessary iff there exists at least one category with more than M tuples in the current level; otherwise, the categorization is complete and the algorithm terminates. If the next level is necessary, let $\mathcal{S}$ denote the set of categories at level *(l-1)* with more than M tuples. Any attribute that has been retained after the attribute elimination step described in Section 5.1.1 and not used as a categorizing attribute in an earlier level is a candidate for the categorizing attribute at this level. For each such candidate attribute A, we partition each category C in $\mathcal{S}$ using the partitioning algorithm described in Sections 5.1.2 and 5.1.3 (depending on whether A is categorical or numeric). Much of the work of partitioning is done just once for the entire level (e.g., sorting the values based on *occ*(v_i) in the categorical case or determining the best splitpoints in the numeric case); only the determination of which subcategories are necessary are done on a per-category basis. We compute the cost of the attribute-partitioning combination for each candidate attribute A and select the attribute α with the minimum cost. For each category C in $\mathcal{S}$, we add the partitions of C based on α to T. This completes the creating of nodes at level *l* after which we move on to the next level.

The above algorithm relies on the assumption that the values the user is interested in for one attribute are *independent* of those she is interested in for another attribute; the quality of the categorization can be improved by weakening this independence assumption and leveraging the correlations captured in the workload. The efficiency of the algorithm can also be improved by avoiding repeated work; we are pursuing such improvements in our ongoing work.

6. EXPERIMENTAL EVALUATION

In this section, we present the results of an extensive empirical study we have conducted to (1) evaluate the accuracy of our cost models in modeling information overload and (2) evaluate our cost-based categorization algorithm and compare it with categorization techniques that do not consider such cost models. Our experiments consist of a real-life user study as well as a novel, large-scale, simulated, cross-validated user-study. The major findings of our study can be summarized as follows:

```
Algorithm CategorizeResults(R)
begin
Create a root ("ALL") node (level = 0) and add to T
$l$ = 1;   // set current level to 1
while there exists at least one category at level $l-1$ with $|tset(C)|>M$
   $\mathcal{S} \leftarrow \{C \mid C$ is a category at level $(l-1)$ and $|tset(C)|>M\}$
   for each attribute A retained and not used so far
      if A is a categorical attribute
         $SCL \leftarrow$ list of single value categories in desc order of $occ(v_i)$
         for each category C in $\mathcal{S}$
            Tree(C,A) $\leftarrow$ Tree with C as root and each non-empty cat
                           C' $\in$ SCL   in same order as children of C
      else // A is a numeric attribute
         SPL $\leftarrow$ list of potential splitpoints sorted by goodness score
         for each category C in $\mathcal{S}$
            Select (m-1) top necessary splitpoints from SPL
            Tree(C,A) $\leftarrow$ Tree with C as root with corr. buckets in
                           ascending order  of values as children of C
      $COST_A \leftarrow \sum_{C \in \mathcal{S}} P(C)*Cost_{All}(Tree(C,A))$
   Select $\alpha = \text{argmin}_A\ COST_A$ as categorizing attribute for level $l$
   for each category C in $\mathcal{S}$
      Add partitioning  Tree(C,$\alpha$)obtained using attribute $\alpha$ to T
   $l = l + 1$; //finished creating nodes at this level, go to next level
end
```

Figure 6: Multilevel Categorization Algorithm

- **Accurate Cost Model**: There is a strong positive correlation between the estimated average exploration cost and actual exploration cost for various users. This indicates that our workload-based cost models can accurately model information overload in real life.
- **Better Categorization Algorithm**: Our cost-based categorization algorithm produces significantly better category trees compared to techniques that do not consider such analytical models.

Thus, our experimental results validate the thesis of this paper that intelligent automatic categorization can reduce the problem of information overload significantly. In the following two subsections, we describe the large-scale, simulated user study and the real-life user study respectively. Both studies were conducted in the context of the home searching application using a real-life home listing database obtained from MSN House&Home. M (max tuples per category) was set to 20 for both studies. All experiments reported in this section were conducted on a Compaq Evo W8000 machine with 1.7GHz CPU and 768MB RAM, running Windows XP.

6.1 Experimental Methodology

Dataset: For both studies, our dataset comprises of a single table called ListProperty which contains information about real homes that are available for sale in the whole of United States. The table ListProperty contains 1.7 million rows (each row is a home) and, in addition to other attributes, has the following non-null attributes: location (neighborhood, city, state, zipcode), price, bedroomcount, bathcount, year-built, property-type (whether it is a single family home or condo etc.) and square-footage.

Workload & Preprocessing: Our workload comprises of 176,262 SQL query strings representing searches conducted by real home buyers on the MSN House&Home web site (tracked over several months). These query strings are selection queries on the ListProperty table with selection conditions on one or more of the attributes listed above. During the preprocessing phase, we scan the workload and build the following tables: the **AttributeUsageCounts** table (as shown in Figure 4(a)), one **OccurrenceCounts** table (as shown in Figure 4(b)) for each potential categorizing attribute that is categorical (viz, neighborhood, property-type) and one **SplitPoints** table (as shown in Figure 5(b)) for each potential categorizing attribute that is numeric (viz, price, bedroomcount, square-footage, year-built). For the numeric attributes, viz. price, square-footage and year-built, the separation interval between the splitpoints was chosen to be 5000, 100 and 5 respectively.

Comparison: In both studies, we compare our cost-based categorization algorithm to two techniques, namely, '*No Cost*' and '*Attr-cost*'. The '*No cost*' technique uses the same level-by-level categorization algorithm shown in Figure 6 but selects the categorizing attribute at each level *arbitrarily* (without replacement) from a predefined set of potential categorizing attributes (the set comprises of neighborhood, property-type, bedroomcount, price, year-built and square-footage). The partitioning based on a categorical attribute simply produces single valued categories in *arbitrary* order while that based on a numeric attribute partitions the range into *equal width buckets* of width 5 times the width of the separation interval (i.e., for price, the range (vmin,vmax] is split at every multiple of 25000; for square footage, at every multiple of 500, etc.). Subsequently, all the empty categories are removed. The '*Attr-cost'* technique selects the attribute with the lowest cost as the categorizing attribute at each level but considers only those partitionings considered by the 'No cost' technique, i.e., arbitrarily ordered, single-valued categories for a categorical attribute and equiwidth buckets for a numeric attribute.

6.2 Large-scale, simulated user-study

Due to the difficulty of conducting a large-scale real-life user study, we develop a novel way to simulate a large scale user study. We pick a subset of 100 queries from the workload and imagine them as *user explorations*, i.e., each workload query W in the subset represents a user who drills down into those categories of the category tree T that satisfy the selection conditions in W and ignores the rest. We refer to a workload query W as a *synthetic exploration*. Since the category tree T must subsume the synthetic exploration W on T, we obtain the user query Q_w (for which T is generated) corresponding to W by *broadening* W. In this study, we broaden W by expanding the set of neighborhoods in W to *all* neighborhoods in the region (e.g., examples of regions are Seattle/Bellevue, NYC – Manhattan, Bronx etc.) and removing all other selection conditions in W; we have tried other broadening strategies and have obtained similar results. For the chosen subset of 100 synthetic explorations, we remove those queries from the workload and build the count tables based on the remaining workload. Subsequently, for each user query Q_w (obtained by broadening W) and for each technique (viz., Cost-based, Attr-cost and No cost), we generate the category tree T based on those count tables, compute the *estimated (average) cost* $Cost_{All}(T)$ of exploration and compute the *actual cost* $Cost_{All}(W,T)$ of exploration (i.e., actual number of items examined by user during the synthetic exploration W using T assuming that she drills down into all categories of T overlapping with W and ignores the rest). For cross validation purposes, we repeat the above procedure for 8 mutually disjoint subsets of 100 synthetic explorations each, each time building the count tables on the remaining workload and

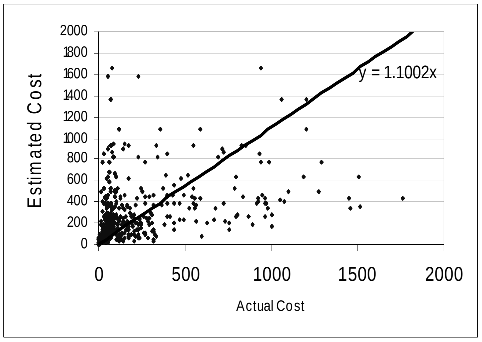


Figure 7: Correlation between actual cost and estimated cost

Subset	Correlation
1	0.39
2	0.98
3	0.32
4	0.48
5	0.16
6	0.16
7	0.19
8	0.76
All	0.90

Table 1: Pearson's Correlation between estimated cost and actual cost

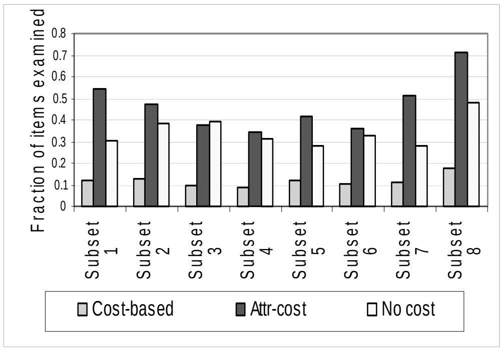


Figure 8: Cost of various techniques

generating the categorizations based on those tables. Figure 7 plots the estimated cost against the actual cost for the 800 explorations. The plot along with the trend line (best linear fit with intercept 0 is y = 1.1002x) shows that the actual cost incurred by (simulated) users has strong positive correlation with the estimated average cost. To further confirm the strength of the positive correlation between the estimated and actual costs, we compute the Pearson Correlation Coefficient for each subset separately as well as together in Table 1.[8] The overall correlation (0.9) indicates almost perfect linear relationship while the subset correlations show either weak (between 0.2 and 0.6) or strong (between 0.6 and 1.0) positive correlation. This shows that our cost models accurately model information overload faced by users in real-life. Figure 8 compares the proposed technique with the Attr-cost and No cost techniques based on $AVG_{W\in subset}$ $Cost_{All}(W,T)/|Result(Q_w)|$, i.e., the fractional cost averaged over the queries in a subset. We consider the fractional cost $Cost_{All}(W,T)/|Result(Q_w)|$ instead of the actual cost $Cost_{All}(W,T)$ to be able to average it across different queries (with different result set sizes) meaningfully. For each subset, the cost-based technique is significantly better (factor of 3-8) compared to the other techniques. Not only did the other techniques produce larger trees, the explorations drilled into more categories due to poor choice of categorizing attributes as well as poor choice of partitions resulting in higher cost. Even though these synthetic explorations touched more tuples than real-life explorations (since the synthetic explorations are actually queries in real-life), users needed to examine less than 10% of the result set when they used cost-based categorization compared to scanning the whole result set (which is the cost if no categorization is used). We expect this percentage to be much lower is real life as confirmed by our real-life user study described in Section 5.2. Surprisingly, Attr-cost is often worse than No cost because the former often produced bigger category trees indicating that cost-based attribute selection is beneficial only when used in conjunction with a cost-based intra-level partitioning.

6.3 Real-life user-study

We conducted a real-life user study with 11 subjects (employees and interns at our organization). We designed 4 search tasks based on familiarity of the subjects with the regions, namely find interesting homes in:

1. Any neighborhood in Seattle/Bellevue, Price < 1 Million
2. Any neighborhood in Bay Area – Penin/SanJose, Price between 300K and 500K
3. 15 selected neighborhoods in NYC – Manhattan, Bronx, Price < 1 Million
4. Any neighborhood in Seattle/Bellevue, Price between 200K and 400K, BedroomCount between 3 and 4.

Our tasks are representative of the typical home searches on the MSN House&Home web site. We evaluated the 3 techniques for each task (which were named 1, 2 and 3 in order to withhold their identities from the subjects); so 12 task-technique combinations in total. We assigned the task-technique combinations to the subjects such that (1) no subject performs any task more than once (2) the techniques are varied across the tasks performed by a subject (so that she can comment on the effectiveness of the techniques) and (3) each task-technique combination is performed by at least 2 (typically more) subjects. For the user study, we built a web-based interface that allows searching the database using a specified categorization technique and renders

User	Correlation
U1	0.73
U2	0.97
U3	0.72
U4	0.66
U5	0.75
U6	0.60
U7	1.00
U8	0.30
U9	-0.08
U10	0.68
U11	0.99
average	0.67

Table 2: Correlation between actual and estimated cost

Task #	Cost-based	No Categorization
1	17.1279	17949
2	10.5481	2597
3	4.62247	574
4	8.01937	7147

Table 3: Comparison of Cost-based categorization to "No Categorization"

[8] The Pearson Coefficient measures the strength of a linear relationship between two variables. The definition as well as a discussion on how to interpret the values can be found at http://www.cmh.edu/stats/definitions/correlation.htm.

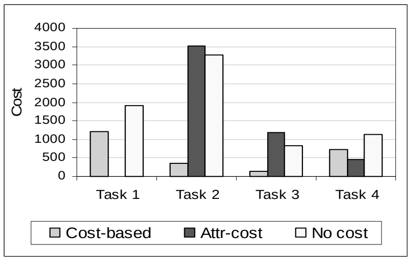


Figure 9: Average cost (#items examined by user till she finds all relevant tuples) of various techniques

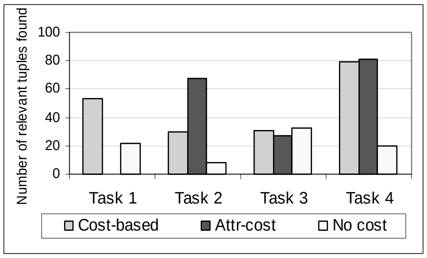


Figure 10: Average number of relevant tuples found by users for the various techniques

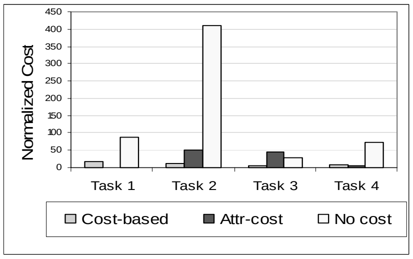


Figure 11: Average normalized cost (#items examined by user per relevant tuple found) of various techniques

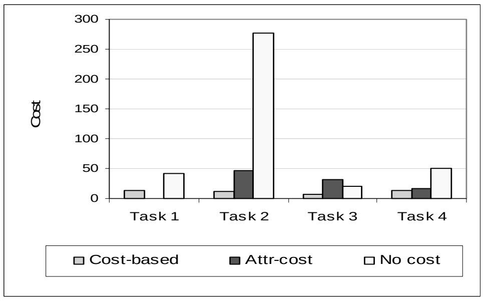


Figure 12: Average cost (#items examined by user till she finds first relevant tuple) of various techniques

Categorization Technique	#subjects that called it best
Cost-based	8
Attr-cost	1
No cost	0
Did not respond	2

Table 4: Results of post-study survey

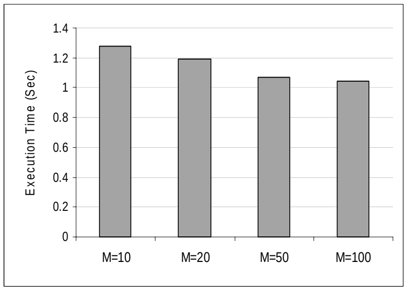


Figure 13: Average execution time of cost-based categorization algorithm

the category tree using a treeview control, all via the web-browser. The treeview allows the user to click on a category to view all items in the category (option SHOWTUPLES) or drill-down/roll-up into categories using expand/collapse operations (option SHOWCAT). We asked the subjects to explore the results using the treeview (exploring, using either SHOWTUPLES or SHOWCAT, only those categories they are interested in and ignoring the rest) till they find *every* relevant tuple. We also asked the subjects to click on the relevant tuples. To compute the measurements reported in this section, we record the following information for each exploration: the subject, the task number, the categorization technique used, the tree generated, the click/expand/collapse operations on the treeview nodes and the clicks on the data tuples along with the timing information. We gave a short demonstration of the system to each subject before the experiment to familiarize them to the system.

Table 2 shows the Pearson's correlation between the estimated average cost $\text{Cost}_{\text{All}}(T)$ and the actual cost (actual number of items examined by the user during the exploration till she found all relevant tuples). On average, there is a strong positive correlation (0.67) between the two costs; in 9 out of the 11 cases, the correlation was strongly positive (between 0.6 and 1.0). This confirms that our cost models accurately model information overload in real life. Figure 9 compares the cost (number of items examined till all relevant tuples found) of the various techniques for each task, averaged across all user explorations for that task-technique combination. For each task, the cost-based technique consistently outperforms the other techniques. We have no results for Task 1-Technique 2 because the tree was very large for this task-technique combination and subjects had problems viewing it on the web browser. The above cost, however, is not the fairest metric for comparison. Unlike in the simulated user study, subjects actually found different number of relevant tuples for the same task when they used different techniques. For a particular categorization technique, if users explored more categories because the categorizations produced by the technique were useful and were helping them to find more relevant tuples, the above metric would unfairly penalize that technique. Figure 10 shows that subjects typically found many more relevant tuples (3-5 times) using the cost-based categorization compared to the no cost technique. This indicates that a good categorization technique not only reduces the effort to find the relevant tuples but also *helps users to find more relevant tuples*. Figure 11 shows a better comparison of the various techniques based on the normalized cost ($\frac{\#\text{ items examined till all relevant tuples found}}{\#\text{ relevant tuples found}}$), averaged across all user explorations for that task-technique combination. The cost-based technique consistently outperforms the no cost technique by one to two orders of magnitude. Using the cost-based technique, subjects typically needed to examine about 5-10 items to find each relevant tuple; that is 3 orders of magnitude less compared to size of the result set (which is the cost if no categorization is used) as shown in Table 3. Figure 12 shows the average cost of the various techniques in the scenario when the user is interested in just one relevant tuple. As in the 'all' case, subjects examined significantly fewer items to find the first relevant tuple using the cost-based technique compared to the other ones. At the end of the study, we asked the subjects which categorization technique worked the best for them among all the tasks they tried. The result of that survey is reported in Table 4. 8 out of 9 users that responded considered technique 1 (which was the cost-based technique) to be the best. Figure 13 shows the execution times of our hierarchical categorization algorithm for various values of M (averaged over 100 queries taken from the workload, average result set size about 2000). Although our algorithm can be further optimized for speed, the current algorithm has about 1 sec response time (including the time to access the count tables in the database which is a significant portion of the response time) for reasonably large sized queries.

7. CONCLUSION

In interactive and exploratory retrieval, queries often return too many answers – a phenomenon referred to as "information overload". In this paper, we proposed automatic categorization of query results to address the above problem. Our solution is to dynamically generate a labeled, hierarchical category structure – the user can determine whether a category is relevant or not by examining simply its label and explore only the relevant categories, thereby reducing information overload. We developed analytical models to estimate information overload faced by users. Based on those models, we developed algorithms to generate the tree that minimizes information overload. We conducted extensive experiments to evaluate our cost models as well as our categorization algorithms. Our experiments show that our cost models accurately model information overload in real-life as there is a strong positive correlation (90% Pearson Correlation) between estimated and actual costs. Furthermore, our cost-based categorization algorithm produces significant better category trees compared to techniques that do not consider such cost-models (one to two orders of magnitude better in terms of information overload). Our user study shows that using our category trees, users needed to examine only about 5-10 items per relevant tuple found which is 3 orders of magnitude less compared to the size of the result set (which is the cost if no categorization is used).

8. ACKNOWLEDGEMENTS

We thank Venky Ganti and Raghav Kaushik for their valuable comments on the paper.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri and G. Das. *DBExplorer: A System for Keyword Search over Relational Databases.* In Proceedings of ICDE Conference, 2002.

[2] S. Agrawal, S. Chaudhuri, G. Das and A. Gionis. *Automated Ranking of Database Query Results.* In Proceedings of First Biennial Conference on Innovative Data Systems Research (CIDR), 2003.

[3] R. Baeza_yates and B. Ribiero-Neto, *Modern Information Retrieval*, ACM Press, 1999.

[4] N. Bruno, L. Gravano and S. Chaudhuri, Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. In ACM TODS, Vo, 27, No. 2, June 2002.

[5] N. Bruno, S. Chaudhuri and L. Gravano. *STHoles: A Multidimensional Workload-Aware Histogram*. Proc. of SIGMOD, 2001.

[6] S. Card, J. MacKinlay and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*, Morgan Kaufmann; 1st edition (1999).

[7] J. Chu-Carroll, J. Prager, Y. Ravin and C. Cesar, A Hybrid Approach to Natural Language Web Search, In Proc. of Conference on Empirical Methods in Natural Language Processing, July 20

[8] S. Dar, G. Entin, S. Geva and E. Palmon, DTL's DataSpot: Database Exploration Using Plain Language, In Proceedings of VLDB Conference, 1998.

[9] S. Dumais, J. Platt, D. Heckerman and M. Sahami, Inductive learning algorithms and representations for text categorization, In Proc. Of CIKM Conference, 1998.

[10] U. Fayyad and K. Irani. *Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning*. Proc. of IJCAI, 1993.

[11] V. Ganti, J. Gehrke and R. Ramakrishnan. *CACTUS - Clustering Categorical Data Using Summaries*. KDD, 1999.

[12] J. Gehrke, V. Ganti, R. Ramakrishnan, W. Loh. *BOAT-Optimistic Decision Tree Construction*. Proc. of SIGMOD, 1999.

[13] 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*. Journal of Data Mining and Knowledge Discovery", Vol 1, No. 1, 1997.

[14] V. Hristidis and Y. Papakonstantinou, DISCOVER: Keyword Search in Relational Databases, In Proc. of VLDB Conference, 2002

[15] V. Poosala, Y. Ioannidis, P. Haas, E. Shekita. *Improved Histograms for Selectivity Estimation of Range Predicates*. Proc. of SIGMOD, 1996.

[16] F. Sebastiani, Machine learning in automated text categorization, ACM Computing Surveys, Vol. 34, No. 1, 2002.

[17] T. Zhang, R. Ramakrishnan and M. Livny. *BIRCH: an efficient data clustering method for very large databases*. Proc. of ACM SIGMOD Conference, 1996.

APPENDIX A

Let us consider 2 different orderings of the same set of *n* subcategories of a non-leaf category C:

Ordering 1: $C_1, C_2, \ldots, C_{i-1}, C_A, C_B, C_{i+1}, \ldots, C_n$

Ordering 2: $C_1, C_2, \ldots, C_{i-1}, C_B, C_A, C_{i+1}, \ldots, C_n$

The two orderings are identical except that the *i*th and *(i+1)*th categories are swapped: while C_A is the *i*th subcategory and C_B the *(i+1)*th subcategory in ordering 1, C_B is the *i*th subcategory and C_A be the *(i+1)*th subcategory in ordering 2. Based on equation 2, Cost_{One} for ordering 1 is: $P(C_1)*\text{Cost}(C_1) + (1-P(C_1)*P(C_2)*\text{Cost}(C_2) + \ldots + \prod_{j=1}^{(i-1)}(1-P(C_i))*P(C_A)*\text{Cost}(C_A) + \prod_{j=1}^{(i-1)}(1-P(C_i))*(1-P(C_A))*P(C_B)*\text{Cost}(C_B) + \ldots$

We consider just the SHOWCAT cost since SHOWTUPLES cost is not affected by the choice of partitioning.

Cost_{One} for ordering 2 is: $P(C_1)*\text{Cost}(C_1) + (1-P(C_1)*P(C_2)*\text{Cost}(C_2) + \ldots + \prod_{j=1}^{(i-1)}(1-P(C_i))*P(C_B)*\text{Cost}(C_B) + \prod_{j=1}^{(i-1)}(1-P(C_i))*(1-P(C_B))*P(C_A)*\text{Cost}(C_A) + \ldots$

Ordering 1 is better than ordering 2 if the Cost_{One} for ordering 1is less than that for ordering 2. Since all terms except the *i*th and *(i+1)*th terms are identical, the first ordering is better if

$$\prod_{j=1}^{(i-1)}(1-P(C_i))*P(C_A)*\text{Cost}(C_A) + \prod_{j=1}^{(i-1)}(1-P(C_i))*(1-P(C_A))*P(C_B)*\text{Cost}(C_B) < \prod_{j=1}^{(i-1)}(1-P(C_i))*P(C_B)*\text{Cost}(C_B) + \prod_{j=1}^{(i-1)}(1-P(C_i))*(1-P(C_B))*P(C_A)*\text{Cost}(C_A)$$

$$\Rightarrow\ P(C_A)*\text{Cost}(C_A) + (1-P(C_A))*P(C_B)*\text{Cost}(C_B) < P(C_B)*\text{Cost}(C_B) + (1-P(C_B))*P(C_A)*\text{Cost}(C_A)$$

$$\Rightarrow\ 1/P(C_A) + \text{Cost}(C_A) < 1/P(C_B) + \text{Cost}(C_B)$$

By induction, it follows that the ordering in increasing values of $1/P(C_i) + \text{Cost}(C_i)$ produces the optimal ordering.