Maximizing Visibility in Skylines

Muhammed Miah
mmiah@suno.edu
Management Information Systems Department
Southern University at New Orleans
New Orleans, LA 70126, USA

Abstract

Given a new product (a tuple), we consider the problem of selecting a small subset of attributes to highlight such that the product stands out in a crowd of existing competitive products and is widely visible to the pool of potential customers. This problem has applications in marketing and product manufacturing and has been the subject of recent investigations. In this paper, we consider an important variant where a product is considered to be visible to a customer if it occurs in the skyline of the query posed by the customer. Given a set of *d*-dimensional points, a skyline query returns points that are not dominated by any other point on all dimensions. This problem variant poses new challenges that cannot be solved optimally using prior techniques. We develop a novel optimal algorithm based on the Signature Tree data structure as well as approximation algorithms to solve the problem. We conduct a performance study illustrating the benefits of our methods on real as well as synthetic data.

Keywords: maximize visibility, subset of attributes, skylines, signature tree, algorithms.

1. INTRODUCTION

Skyline query processing has been extensively investigated in recent years. Given a set of points, the skyline comprises of the points that are not dominated by other points. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension. For example, a student attending a conference might want to search within a hotels database for a cheap hotel with reasonable ratings near the conference venue. This kind of query sometimes contains conflicting goals, as hotels near the conference venue with reasonable ratings are expected to be rather expensive. It is of interest to return as query results the set of skyline hotels; where for each skyline hotel there is no other hotel that is cheaper, nearer, and with better ratings. While skylines are naturally defined for numeric data, they can also be defined for categorical and Boolean data if the data values within each attribute's domain have natural total (or even partial) orderings. In this paper we mainly consider Boolean skylines (skylines with Boolean data), where all the attributes asked by a query need not to be present in the tuple to be returned by the query. For example, let us consider a car database with Boolean attributes such as whether the car has *AC*, *Power Doors*, *Power Brakes*, etc. Thus if a user poses a query such as "Select * from Cars where Make = Honda and AC = yes and Power Windows = yes", then a car such as <*Toyota, AC, Power Windows*> would appear in the skyline if there is no car that exactly satisfies the query conditions.

Thus, skyline query processing techniques or skyline operators are designed to provide all interesting answers that may satisfy a user's need. Skyline semantics are less rigid than conjunctive range query semantics and can be of use in exploratory search applications.

©2011 EDSIG (Education Special Interest Group of the AITP)

Our goal in this paper is however not to design a skyline operator. Instead, we consider an interesting generalization of a problem that has applications in marketing and product manufacturing, and has been the subject of recent investigations (Miah, Das, Hristidis, & Mannila, 2009). Given a new product (a tuple), we consider the problem of selecting a small subset of attributes to highlight such that the product stands out in a crowd of existing competitive products and is widely visible to the pool of potential customers. So the goal is not to develop a better search technique to help the user (buyer of a product) but to help the seller of the product to reach maximum number of users. This problem was investigated by Miah et al. (2009), where primarily a somewhat rigid model of conjunctive query semantics was used to define product visibility – a product is visible to a customer if it satisfies all conditions of the query posed by the customer. The skyline variant of the problem was also discussed very briefly and the same solution was proposed for both the Boolean and skyline problem variants, which later proved not to be optimal for skyline variant (Miah, 2009). In this paper we mainly consider skyline semantics – a product is visible to a customer if it appears in the skyline of the customer's query, i.e., although it may not exactly match all query conditions, it is nevertheless potentially "more interesting" to the customer than many other competing products.

Selecting [a/the] subset of attributes to highlight a product plays an important role in marketing and manufacturing of products as well as in operation[s] research. We can consider a real world scenario: assume that one wishes to publish a classified-ad in a newspaper (online or printed) to advertise a house for sale. The house may have a lot of attributes (number of bedrooms, bathrooms, close to beach, close to school, etc.). However, due to the advertisement costs involved, it is not possible to describe all attributes in the ad. So one has to select, say the ten best attributes. Which ones should be selected within the budget limit such that the published ad will be viewed by as many customers as possible? From a manufacturing point of view, a house builder might want to find the most popular combination of features to add to a house to be constructed in so as to be more interesting than other competing homes to as many potential customers as possible. Another interesting real world example would be to select a small set of keywords or a title for an advertisement campaign.

In this paper, we mainly focus on the important and interesting variant of the problem where the data is Boolean and the queries follow skyline retrieval semantics. Here, a tuple does not have to have all the attributes present asked by a query, but it has to be visible on the skyline of the query. This problem variant cannot be solved optimally by the existing algorithms. Moreover one query can have multiple skylines, i.e., multiple sets of attributes for which different data points (tuples) are visible on the skyline of the query. We develop a technique to solve the skyline version of the problem that is quite different from the methods proposed by Miah et al. (2009). The new method is based on a judicious application of the signature tree data structure (Chen & Chen, 2006 and Miah, 2009) with modifications and smart prunings. Interestingly, our new algorithm can also solve easily the old problem variant of conjunctive query semantics optimally.

Our main contributions are summarized below:

1. We investigate the problem of selecting attributes of a tuple for maximum visibility in skylines as a promising data exploration problem that benefits a certain class of users interested in designing and marketing their products.
2. Though the problem is proved to be NP-hard (Miah et al., 2009), we are able to develop an optimal algorithm based on the signature tree data structure to solve the problem that works well for moderate problem instances.
3. We also present fast approximation algorithms that work well for larger problem instances.
4. We perform detailed performance evaluations on both real and synthetic data to demonstrate the effectiveness of our developed algorithms.

2. PROBLEM FRAMEWORK

Before formally defining the problem, we first provide some useful definitions and notations in Appendix 1.

The problem formally can be defined as follows.

PROBLEM: *Given a database of competing products D, a query log Q with Skyline Query semantics, a new tuple t, and an integer m,*

compute a compressed tuple t′ by retaining m attributes such that the number of queries that retrieve t′ on the skylines is maximized.

Car ID	*Attributes/Features present in the car*
t_1	{AC, Four Door, Power Doors}
t_2	{Four Door, Turbo}
t_3	{AC, Auto Trans, Power Brakes, Power Doors}
t_4	{AC, Four Door, Power Brakes, Power Doors}
t_5	{AC, Four Door}
t_6	{Four Door, Power Doors}
t_7	{Power Doors, Turbo}

Database *D*

Query ID	*Attributes/Features asked by the query*
q_1	{AC, Four Door}
q_2	{AC, Power Doors}
q_3	{Four Door, Power Doors}
q_4	{Power Brakes, Power Doors}
q_5	{Auto Trans, Turbo}

Query Log *Q*

New Car	*Attributes/Features present in the car*
t	{AC, Auto Trans, Four Door, Power Brakes, Power Doors}

New tuple *t* to be inserted

Figure 2. Running EXAMPLE 1

The following running example will be used to illustrate problem.

EXAMPLE 1: *Consider an inventory database of an auto dealer, which contains a single database table D with N=7 rows and M=6 possible attributes a car can have (AC, Auto Trans, Four Door, Power Brakes, Power Doors, and Turbo) where each tuple represents a car for sale. The table has numerous attributes that describe details of the car: Boolean attributes such as AC, Four Door, etc; categorical attributes such as Make, Color, etc; numeric attributes such as Price, Age, etc; and text attributes such as Reviews, Accident History, and so on. Figure 2 illustrates such a database (where only the Boolean attributes are shown) of seven cars already advertised for sale. The figure also illustrates a query log of five queries, and a new car t that needs to be advertised, i.e., inserted into this database.* □

Now we can find the skyline points (cars) for the given database *D* and Query log *Q* in Figure 2. As we know, a skyline point is a point which is not dominated by any other point in all dimensions. For query q_1 {*AC, Four Door*}, we can see it easily that tuples t_4 and t_5 are the tuples (skyline points) which are not dominated by any other tuples in *D*. For query q_2 {*AC, Power Doors*}, tuples t_3 and t_4 are the skyline points, and so on. Table 1 (Appendix 2) displays all the skylines found for the given query log *Q* and database *D*. A skyline tuple or data points can have many attributes but we are interested only in the attributes for which the tuple is visible on the skyline, as our goal is to find the subset of attributes for the new tuple which will maximize the number of queries having the new tuple visible on their skylines. A query can have more than one skyline; e.g., for query q_5, tuples t_2 and t_7 are visible on one skyline whereas tuple t_3 is visible on another skyline. We keep separate record for each skyline as shown in Table 1 (Appendix 2).

Suppose we are required to retain $m = 3$ attributes of the new tuple. It is not hard to see that if we retain the attributes *AC*, *Four Door*, and *Power Doors* (i.e., $t' = \{AC, Four\ Door, Power\ Doors\}$), the compressed tuple *t′* will be visible on the skylines for the maximum of three queries (q_1, q_2, and q_3). No other selection of three attributes of the new tuple will remain on skylines of more queries.

3. OPTIMAL ALGORITHM

There are several methods proposed for efficient processing of skyline queries such as Block-Nested-Loops and Divide & Conquer (Kossmann & Stocker, 2001), Bitmap-based and Index-based (Tan et al. 2001), Nearest Neighbor (Kossmann, Ramsak, & Rost, 2002), Branch and Bound Skyline (Papadias, Tao, Fu, & Seeger, 2003). Any good skyline processing technique can be used here to find the skylines for the query log. We can assume that the skylines for each query in the log have already been computed by any one of these algorithms. Once these skylines have been found, then our problem is to find the subset of the attributes for the new tuple so that skylines from the

maximum number of queries will retrieve the new tuple.

A Naïve optimal algorithm and its infeasibility are discussed in Appendix 3.

We propose a novel optimal algorithm based on Signature Tree data structure (Chen & Chen, 2006) which is much more efficient than the Naïve algorithm.

Optimal Algorithm Based on Signature Tree (*AST*)

We adapt the candidate set generating function *apriori-gen* used in the *Apriori* algorithm for mining association rules (Agrawal & Srikant, 1994) to generate possible candidate sets in this algorithm. The *apriori-gen* function takes L_{k-1}, the set of all large (k-1) itemsets. It returns a superset of set of all large k-itemsets. The function works as shown in Figure 3.

First in the *join* step, it joins L_{k-1} with L_{k-1}:

Insert into C_k
select $p.\text{item}_1$, $p.\text{item}_2$, …, $p.\text{item}_{k-1}$, $q.\text{item}_{k-1}$
from L_{k-1} p, L_{k-1} q
where $p.\text{item}_1 = q.\text{item}_1$, …, $p.\text{item}_{k-2} = q.\text{item}_{k-2}$,
$p.\text{item}_{k-1} = q.\text{item}_{k-1}$

Next, in the *prune* step, it deletes all itemsets $c \in C_k$ such that some (k-1) subset of c is not in L_{k-1}:

for all itemsets $c \in C_k$ do
for all (k-1)-subsets u of c do
if ($u \notin L_{k-1}$ then
delete c from C_k

Figure 3. Function *apriori-gen*

The signature tree, its construction, and definition are discussed in Appendix 4. We build a balanced signature tree for the skylines using the weight based method (Chen & Chen, 2006). The process of creating the balanced signature tree is discussed in Appendix 4.

The traditional approach of searching the signature tree is discussed in Appendix 5.

The traditional searching of signature tree has two major problems (discussed in Appendix 6). So we propose a new approach for searching the signature tree.

New Approach for Searching the Signature Tree: First we create the signature tree for the skylines as described above. Then, we search the tree at each level from *2*-attributes candidate sets to up to *m*-attributes candidate sets. Candidate sets at each level k (= 2…m) are generated using function *apriori-gen* as discussed above. At each level searching is done as follows:

i. Let v be the node encountered and $s_t[i]$ be the position to be checked.
ii. We move both the right and left child of v whether $s_t[i] = 0$ or $s_t[i] = 1$.
iii. We maintain a variable $m' = (m - k)$, which is the number of *mistakes* allowed. Here, m is the number of attributes we need to retain for the new tuple t, and k is the current number of attributes in the candidate set. A mistake during the search occurs when we move to the right of a node (i.e., skyline has value 1 for the digit mentioned by the node) and the candidate set has value 0 for that digit. If this situation happens, we increase the count for mistakes. Consider the signature tree in Figure 5 (Appendix 4) for our running example. Assume current value of $k = 2$, $m = 3$, and we have a candidate set with signature 110000. So, the value of $m' = m - k = 1$. As mentioned above, when we search the tree, we move both right and left of a node. Moving left never increases the number of mistakes because if a skyline s has value 0 for a digit then a candidate set c can have either value 0 or 1 for the corresponding digit. When we move right from the root node (Figure 5 in Appendix 4), the value of digit 5 (root node) in candidate set 110000 is 0, so we increase the number of mistakes made so far, which is (0+1) = 1. Next we move both left and right from node labeled 1 (right child of root). Moving right from node labeled 1 does not increase the number of mistakes made as the value of digit 1 in candidate set is also 1. But when we move right of the node labeled 3 (left child of node labeled 1), we increase the number of mistakes made because value of digit 3 in the candidate set is 0. Here a new value for the number of mistakes made so far is (1+1) = 2 which is greater than m'. So we do not consider any nodes

to the right of node labeled 3. As we can see from the tree in Figure 5 (Appendix 4), we do not consider s_3 with signature 001010 as a possible subset of the candidate set 110000 in future. We can see easily that adding 1 to the candidate set 110000 will not make s_3 (001010) a subset of the candidate in future.

iv. Once we reach a node and number of mistakes made so far reaching the node from the root is greater than *m'*, then we do not consider the node and its children (if it is an internal node) as the possible subset of the candidate set.

v. Once we reach a leaf node (skyline) and the number of mistakes made so far reaching the node from the root is not greater than *m'*, we keep the skyline for further match.

Once we find the corresponding skylines *S* (leaf nodes) by searching the tree for a candidate set *c*, for each skyline s_i we do the following:

a) We find the number attributes *r* present in skyline s_i which is not present in candidate set *c*. If $r \leq (m - k)$, we count s_i as the possible subset of *c*. Here, *m* is the number of attributes we need to retain for the new tuple *t*, and *k* is the current number of attributes in the candidate set. If $r > (m - k)$, we do not consider that skyline or leaf node for *c* as a possible subset, i.e., do not increase count for *c*.

b) We keep a count for each candidate set *c* that how many skylines have been found as possible subsets of *c*. Here, we count each query only once. For example, considering our running example, if we find two skylines s_5 and s_6 are the subsets of any candidate set c_i, we only count them once because both come from the same query q_5. This is why we keep information in the tree for both the skyline and the query from which it came. We remove the candidate set *c* if the total count for it is less than the *minimum support*.

c) At level *m* (candidate sets with *m*-attributes), we simply check how many skylines (found after searching the tree) are actually the subsets of the candidate set. Again, we count skylines for each query only once for a candidate set. We return the candidate set with highest count as the top-*m* attributes for the new tuple *t*.

4. APPROXIMATION ALGORITHMS

A simple greedy heuristic would be to select the top-*m* attributes with highest frequency in the skyline log (frequency is the number of times an attribute appear in the skyline log). But this is not a good approach when attributes are correlated, which is quite common in practice. We propose three effective approximation algorithms based on greedy heuristics that perform well.

Backward Elimination (*BE*)

We propose a backward elimination greedy heuristic where all single attributes are considered first and then remove one at a time until *m* attributes are left. The summary of the approach is shown in Figure 6 below.

1. Take the original set of attributes, S_o.
2. Remove an attribute randomly from S_o which was not tested (removed) before and count how many skylines are subsets of the new set (S_o -1).
3. Restore the attribute removed in *step 2*.
4. Repeat *steps 2* and *3* for each attribute in S_o.
5. Remove the attribute from S_o permanently with highest count, i.e., remove the attribute which has lowest impact.
6. Repeat *steps 2-5* until S_o remains with *m* attributes

Figure 6. Approximation Algorithm: Backward Elimination (*BE*)

1. Let S_o = set of all original attributes present in skyline log
2. Let *S* = an empty set, and an integer $k = 0$ which is the number of attributes currently present in *S*.
3. For ($k = 0$ to *m*)
4. For each attribute a_i left in S_o, check if we add (m-k) attributes from S_o including a_i to *S*, then how many skylines could be possible subsets of *S*. In fact we check the number of attributes in a skyline which are not present in *S* including a_i. If the number is less than (m – number of attributes in *S* including a_i), then the skyline is considered as a possible subset of the candidate set *S*
5. Add the attribute to *S* with highest count in *step 4*.
6. Remove the attribute added to *S* in *step 5* from S_o.
7. End
8. Return *S*.

Figure 7. Approximation Algorithm: Forward Selection (*FS*)

Forward Selection (*FS*)

Forward selection heuristic starts with an empty set of attributes, *S* and adds one attribute at a time until *S* has *m* attributes. In order to find the best *m* attributes, we can first add the attribute with highest frequency in the skyline log (frequency means number of times the attributes appear in the skyline log). Next we add the attribute which occurs most with the first attribute, then add the attribute which occurs most with the first and second attributes together, and so on until *S* remains with *m* attributes. In this method, we might find *S* is a good selection of *m* attributes if we want to find the new *t* as a subset of maximum number of skylines or queries. But our goal is the opposite; we want to find *S* as a superset of the maximum number of skylines. So, just adding top-*m* attributes may not result a good selection of attributes. We modify the addition criteria of an attribute to *S*. Figure 7 shows the summary of the algorithm *FS*.

Combination of Forward Selection and Backward Elimination (*FSBE*)

Now we propose another heuristic which combines both the algorithms *BE* and *FS* considering bidirectional hill climbing techniques. Hill climbing is a well known procedure for sequential attribute selection. Greedy algorithms such as *BE* and *FS* implement so called unidirectional hill climbing, i.e., attributes once added (removed) cannot be later deleted (added). The advantage of bidirectional hill climbing compared to either *FS* or *BE* is that one or several previously deleted (added) attributes can be brought back to (removed from) the subset if the accuracy of the algorithm increases. But this technique can be time consuming as both the *BE* and *FS* has to perform completely and then somehow combine the results. So, we propose a new technique to improve the performance of the algorithm, described as follows:

Once an attribute is removed by *BE* it is not considered to be added by *FS* anymore. Similarly, once an attribute is added by *FS* it is not considered to be removed by *BE*. So, at every step *BE* eliminates one attribute and *FS* adds one. We repeat the procedure until *FS* adds *m* attributes. The summary of the algorithm *FSBE* is given in Figure 8.

1. Let S_o = set of all original attributes present in skyline log
2. Let S = an empty set, and an integer $k = 0$ which is the number of attributes currently present in S.
3. For ($k = 0$ to m)
4. Perform *BE* on S_o. // every step removes one attribute from S_o.
5. Perform *BS* on S_o. // every step adds one attribute to S.
6. Remove the attribute from S_o which is added by *FS* to S in *step 5*
7. End
8. Return S.

Figure 8. Approximation Algorithm: *FSBE*

5. EXPERIMENTS

In this section we measure (a) the time cost of the proposed optimal and approximation algorithms, and (b) the quality of the approximation algorithms.

The system configuration and details of datasets are discussed in Appendix 7.

The top-*m* attributes selected by our algorithms seem very effective. For example, both for real and synthetic query logs, our optimal algorithm could select top features or attributes specific to a car, e.g., sporty features are selected for sports cars, safety features are selected for passenger sedans, and so on.

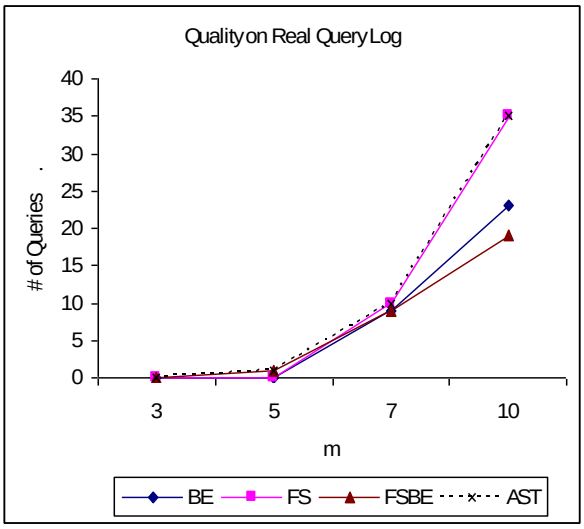

Figure 9. Quality on Real Query Log for varying *m*

Figure 9 shows the quality of the algorithms on the real query log which has a total of 185 queries. The *x*-axis represents *m* which is the number of attributes needing to be selected, and *y*-axis represents the number of queries for which the new tuple with selected *m* attributes is visible on the skylines. We use several experiments for each algorithm with varying *m*. The real query log in fact has no query as well as skyline with less than or equal to 3 attributes, so we can see all algorithms produce zero output for *m* = 3. We can see from the graph that approximation algorithms work really well.

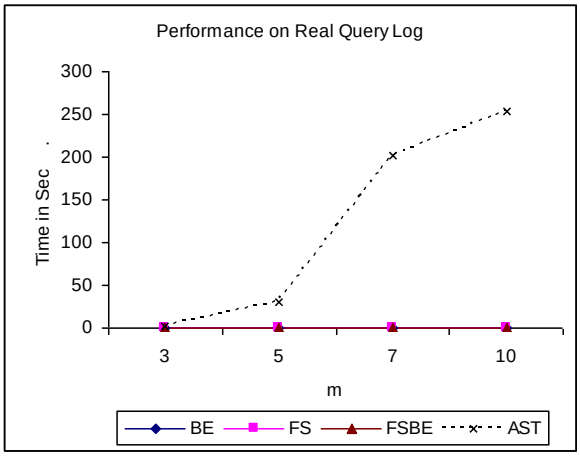

Figure 10. Performance on Real Query Log for varying *m*

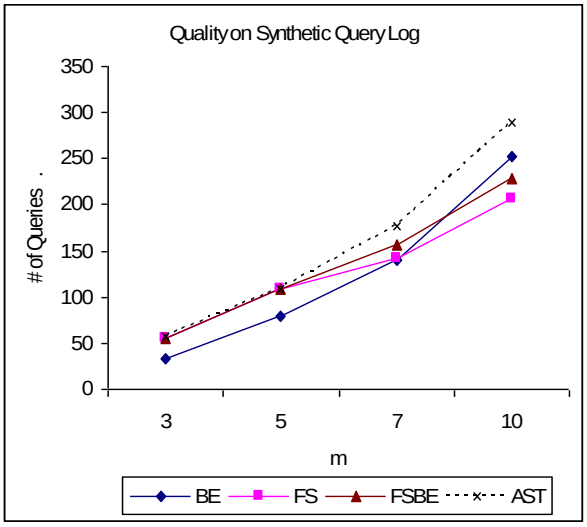

Figure 11. Quality on Synthetic Query Log (1000 queries) for varying *m*

Figure 10 displays the execution times of each algorithm for the real query log. Here *x*-axis represents *m*. The *y*-axis represents the time in seconds to execute the algorithm. We can see that the approximation algorithms are faster than optimal *AST*, which is expected.

Next we show the quality on synthetic query log of size 1000 queries in Figure 11. As we can see from the graph, for the synthetic query log our approximation algorithms also produce very good outputs similar to the real query log.

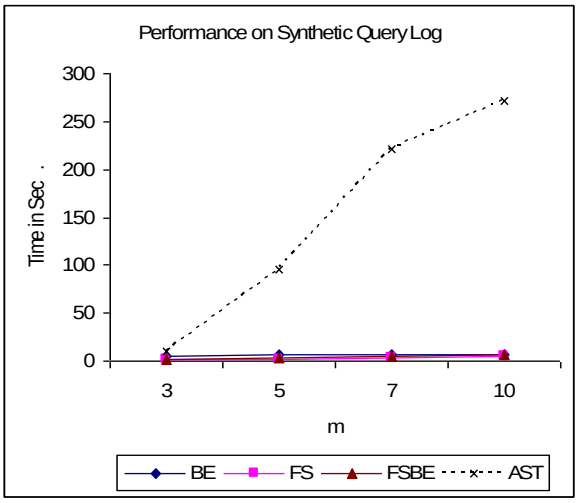

Figure 12. Performance on Synthetic Query Log (1000 queries) for varying *m*

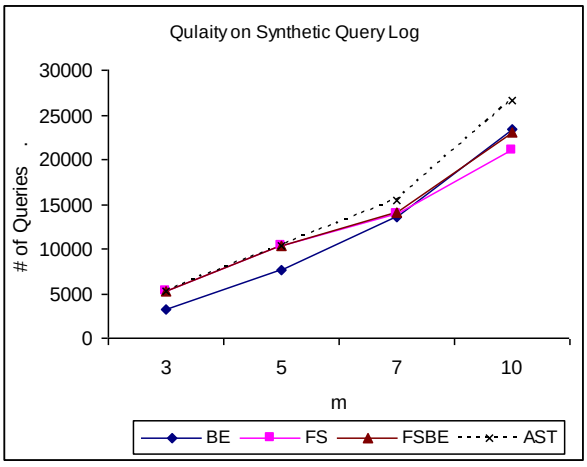

Figure 13. Quality on Synthetic Query Log (100K queries) for varying *m*

Figure 12 shows the execution times of the algorithms for the synthetic query log of 1000 queries. As we can see from the graph, the approximation algorithms are also very fast for synthetic query log. When *m* increases, execution time for *AST* also increases more than approximation algorithms.

Figure 13 and Figure 14 show the quality and execution times respectively of each algorithm for the synthetic query log of 100000 queries. The algorithms perform similar way as in the previous cases.

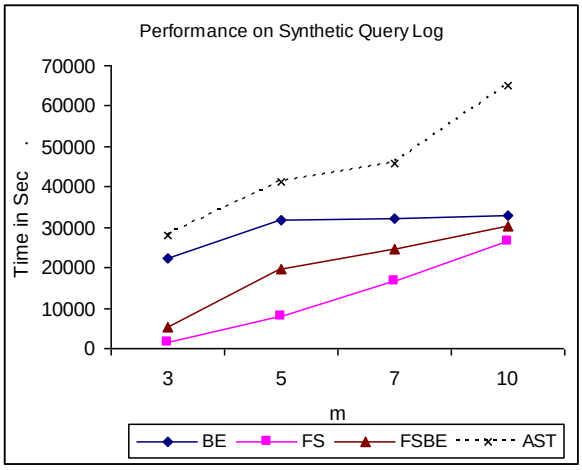

Figure 14. Performance on Synthetic Query Log (100K queries) for varying *m*

We can see from the graphs that approximation algorithm *FS* is faster than both *BE* and *FSBE*, which makes sense. *FS* starts with an empty attribute set and at each step adds one attribute until *m* attributes are added to the set. Usually *m* is a small number compared to the total number of attributes, *M* present in the database. So, *FS* has to iterate only *m* times. On the other hand, algorithm *BE* starts with a set of all *M* attributes and at each step eliminates one attribute from the set until it is left with *m* attributes. As we said, typically *M* is a larger number than *m*, so *BE* has to iterate (*M*-*m*) times which is typically much larger than *m*. So *BE* is slower than *FS*. Because *BE* has to iterate more times, it usually produces better output than *FS* that we can see easily for the synthetic query logs. *FSBE* is the combination of *FS* and *BE*, so it is slower than *FS* but faster than *BE*. But as *FSBE* considers both elimination and addition in each step, it usually produces better output than both *BE* and *FS*.

6. RELATED WORK

A large corpus of work has tackled the problem of ranking the results of a query. In the documents world, the most popular techniques are tf-idf based (Salton, 1989) ranking functions, like BM25 (Robertson & Walker, 1994), as well as link-structure-based techniques like PageRank (Brin & Page, 1998) if such links are present (e.g., the Web). In the database world, automatic ranking techniques for the results of structured queries have been recently proposed ([Agrawal, Chaudhuri, Das, & Gionis, 2003], [Chaudhuri, Das, Hristidis, & Weikum, 2004], [Su, Wang, Huang, & Lochovsky, 2006]). In addition to ranking the results of a query, there has been recent work (Das, Hristidis, Kapoor, & Sudarshan, 2006) on ordering the displayed attributes of query results.

Both of these tuple and the attribute ranking techniques are inapplicable to our problem. The former inputs a database and a query, and outputs a list of database tuples according to a ranking function, and the latter inputs the list of database results and selects a set of attributes that "explain" these results. In contrast, our problem inputs a database, a query log, and a new tuple, and computes a set of attributes that will rank the tuple high for the skylines of as many queries in the query log as possible.

Our work differs from the extensive body of work on *feature selection* (Guyon, & Elisseeff, 2003) because our goal is very specific – to enable a tuple to be highly visible to the users of the database as well as stand out in the crowd of existing products – and not to reduce the cost of building a mining model such as classification or clustering.

Kleinberg, Papadimitriou, & Raghavan (1998) present a set of microeconomic problems suitable for data mining techniques; however no specific solutions are presented. Their problem closer to our work is identifying the best parameters for a marketing strategy in order to maximize the attracted customers, given that the competitor independently also prepares a similar strategy. Our problem is different since we know the competition (other data items). Another area where boosting an item's rank has received attention is Web search, where the most popular techniques involve manipulating the link-structure of the Web to achieve higher visibility (Gori & Witten, 2005).

Computing frequent itemsets is a popular area of research in data mining and some of the best known algorithms include Apriori (Agrawal & Srikant, 1994) and FP-Tree (Han, Pei, & Yin, 2000). In frequent itemset mining, a subset of items are predicted which are frequent (occurs together more than a threshold) in the transaction database. Here, a frequent itemset is basically a subset of a transaction. Our problem is the opposite, we want to identify the subset of attributes (items) which to retain for the new tuple *t* such that *t* becomes a superset of a skyline (transaction).

The works on dominant relationship analysis (Li, Ooi, Tung, & Wang, 2006) and dominating neighborhood profitably (Li, Tung, Jin, & Ester, 2007) are related to our work. The former tries to find out the dominant relationship between products and potential buyers where by analyzing such relationships, companies can position their products more effectively while remaining profitable, and the latter introduces skyline query types taking into account not only min/max attributes (e.g., price, weight) but also spatial attributes (e.g., location attributes) and the relationships between these different attribute types. Their work aims at helping manufacturers choose the right specs for a new product, whereas our work aims at choosing the attributes subset of an existing product for advertising purposes.

Skyline query processing has been well investigated recently. Several techniques have been proposed for efficient skyline query processing ([Borzsonyi, Kossmann, & Stocker, 2001], [Tan, Eng, & Ooi, 2001], [Kossmann, Ramsak, & Rost, 2002], [Papadias, Tao, Fu, & Seeger, 2003]). There has been recent work on categorical skylines (Sarkas, Das, Koudas, & Tung, 2008), where the authors proposed a method for maintaining efficiently the skylines of streaming data with partially ordered, categorical attributes. One main difference of our work with the existing works is that we consider Boolean skylines and our goal is not to propose a method to efficiently process or maintain the skylines, instead we use skylines as a query semantic where a new tuple can be visible for a maximum number of queries.

7. CONCLUSIONS

In this paper we consider a problem that has applications in marketing and product design - given a new product (a tuple), to select a small subset of attributes to highlight such that the product stands out in a crowd of existing competitive products and is widely visible to the pool of potential customers. A product is considered to be visible to a customer if it occurs in the skyline of the query posed by the customer. This problem variant poses new challenges that cannot be solved optimally using prior techniques, hence we develop novel optimal algorithm based on the signature tree data structure as well as approximate algorithms to solve the problem.

Clearly, the definition of visibility of a product to customers can be extended beyond the concept of skylines. As future work we are considering other interesting definitions of product visibility, and investigating whether signature trees and similar techniques can be used for solving such problems.

8. REFERENCES

Agrawal, S., Chaudhuri, S., Das, G., & Gionis, A. (2003). Automated Ranking of Database Query Results. *CIDR* 2003.

Agrawal, R., & Srikant, R. (1994). Fast Algorithms for Mining Association Rules. VLDB 1994: 487-499

Borzsonyi, S., Kossmann, D., & Stocker, K. (2001). The Skyline Operator. *ICDE* 2001.

Brin, S., & Page, L. (1998). The Anatomy of a Large-Scale Hypertextual Web Search Engine. *WWW Conference*, 1998

Chaudhuri, S., Das, G., Hristidis, V., & Weikum, G. (2004). Probabilistic Ranking of Database Query Results. *VLDB* 2004

Chen, Y., & Chen, Y. (2006). On the Signature Tree Construction and Analysis. *IEEE Trans. Knowl. Data Eng.* 18(9): 1207-1224

Das, G., Hristidis, V., Kapoor, N., & Sudarshan, S. (2006). Ordering the Attributes of Query Results. *SIGMOD* 2006.

Guyon, I., & Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3(mar):1157–1182

Gori, M., & Witten, I. (2005). The bubble of web visibility. *Commun. ACM* 48, 3 (Mar. 2005), 115-117

Han, J., Pei, J., & Yin, Y. (2000): Mining Frequent Patterns without Candidate Generation. *SIGMOD* 2000: 1-12.

Kleinberg, J., Papadimitriou, C., & Raghavan, P. (1998). A Microeconomic View of Data Mining. *Data Min. Knowl. Discov*. 2, 4 (Dec. 1998), 311-324

Kossmann, D., Ramsak, F., & Rost, S. (2002). Shooting Stars in the Sky: an Online Algorithm for Skyline Queries. *VLDB* 2002.

Li, C., Tung, A. K. H., Jin, W., & Ester, M. (2007). On Dominating Your Neighborhood Profitably. VLDB 2007: 818-829

Li, C., Ooi, B. C., Tung, A. K. H., & Wang, S. (2006). DADA: a Data Cube for Dominant Relationship Analysis. SIGMOD Conference 2006: 659-670

Miah, M. (2009). An Optimal Signature-Tree based Algorithm for Selecting Attributes for Maximum Visibility. *International Conference on Information Technology (ICIT)* 2009.

Miah, M., Das, G., Hristidis, V., & Mannila, H. (2009). Determining Attributes to Maximize Visibility of Objects. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 2009, vol. 21 no. 7, pp. 959-973.

Papadias, D., Tao, Y., Fu, G., & Seeger, B. (2003). An Optimal and Progressive Algorithm for Skyline Queries. *ACM SIGMOD* 2003

Robertson, S. E., Walker, S. (1994). Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. *SIGIR* 1994

Salton, G. (1989). Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer. *Addison Wesley*, 1989

Sarkas, N., Das, G., Koudas, N., & Tung, A. K. H. (2008). Categorical skylines for streaming data. *SIGMOD Conference* 2008: 239-250

Su, W., Wang, J., Huang, Q., & Lochovsky, F. (2006). Query Result Ranking over E-commerce Web Databases. *ACM CIKM* 2006

Tan, K., Eng, P., & Ooi, B. C. (2001): Efficient Progressive Skyline Computation. *VLDB* 2001.

©2011 EDSIG (Education Special Interest Group of the AITP)

Appendices

Appendix 1: Some Useful Definitions and Notations

***Database*:** Let $D = \{t_1 \ldots t_N\}$ be a collection of tuples with Boolean attributes over the attribute set $A = \{a_1 \ldots a_M\}$, where each tuple t is a set of attributes. Considering a car database, a t has the actual attribute names which represents that an attribute is present in the tuple (e.g., *AC*).

***Tuple Compression*:** Let t be a tuple and let t' be a subset of t with m attributes. Thus t' represents a compressed representation of t.

Query: We view each query as a subset of attributes where users search for a product specifying their attributes of interest.

***Query Log*:** Let $Q = \{q_1 \ldots q_R\}$ be collection of queries where each query q defines a subset of attributes.

***Skyline*:** Given a set of points, the skyline comprises the points that are not dominated by other points. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension (Tan, Eng, & Ooi, 2001). Consider a common example in the literature, "*choosing a set of hotels that is closer to the beach and cheaper than any other hotel in distance and price attributes respectively from the database system of the travel agents'* (Kossmann & Stocker, 2001)". Figure 1 illustrates this case in *2*-dimensional space, where each point corresponds to a hotel record. The *x*-axis specifies the room price of a hotel, and the *y*-axis specifies its distance to the beach. Clearly, the most interesting hotels are the ones *{a*, *g*, *i*, *n}*, called *skyline*, for which there is not any other hotel in *{a*, *b*, . . . , *m*, *n}* that is better on both dimensions.

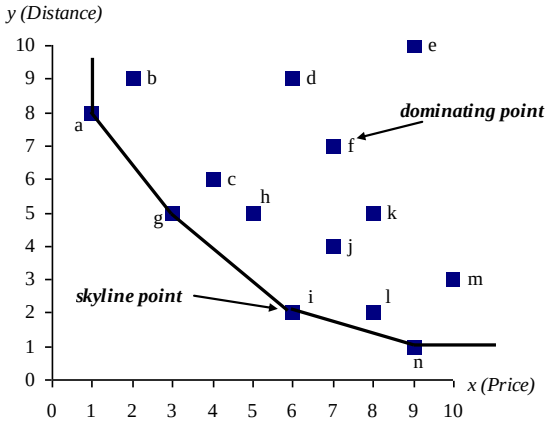

Figure 1. Skyline Example

***Skyline Log*:** Let $S = \{s_1 \ldots s_L\}$ be collection of skylines where each skyline s defines a subset (i.e., projection) of attributes for which any data point (tuple) remains on the skyline. For example, if a user poses a query q = "Select * from Cars where Make = Honda and AC = yes and Power Windows = yes", and the database has three cars t_1 = <*Toyota, AC, Power Windows*>, t_2 = <*Honda, AC, Power Brakes*> and t_3 = <*Nissan, AC, Power Brakes*>. We can see it easily from the skyline definition that the cars t_1 and t_2 will be on the skyline of q, which are not dominated by any other cars (t_3 here) present in the database based on the attributes asked by the query q. We do not store the actual skyline data points (all attributes present in the tuple) such as t_1 and t_3 in skyline log, instead the set of attributes for which a data point is visible on the skyline. Here, t_1 = <*Toyota, AC, Power Windows*> is visible on the skyline of q because it has attributes {*AC, Power Windows*} present asked by q. So,

the skyline we define here as s_1 = {*AC, Power Windows*}. Similarly skyline of for t_2 is s_2 = {*Honda, AC*} for which t_2 is on the skyline of q. Skylines log contains all such skylines for the query log.

Appendix 2: Skylines of queries

Skyline ID	*Query ID*	*Car ID (cars on the skyline)*	*Attributes for which the car is on the skyline*
s_1	q_1	t_4, t_5	{AC, Four Door}
s_2	q_2	t_3, t_4	{AC, Power Doors}
s_3	q_3	t_1, t_4, t_6	{Four Door, Power Doors}
s_4	q_4	t_3, t_4	{Power Brakes, Power Doors}
s_5	q_5	t_2, t_7	{Turbo}
s_6	q_5	t_3	{Auto Trans}

Table 1. Skylines of the Queries

Appendix 3: Optimal Naive Algorithm

The problem is proved to be NP-hard (Miah et al. 2009). A naïve optimal approach to find the subset of attributes (*m* attributes) to retain for the new tuple *t* to maximize the number of queries which will have *t* on the skylines as follows:

I. Generate all possible *m*-attributes candidate sets.
II. For each candidate set *c* in step (I), scan the skyline log and find for how many queries the skylines are the subsets of *c*.
III. Return the candidate set *c* with the highest count.

In practice, the Naïve algorithm is not feasible when the number of attributes is large since the algorithm has to generate a huge number of possible candidate sets. If the database has a total of *M* attributes and we want to retain *m* attributes, then there are total $\binom{M}{m}$ possible attribute sets which can be a very large number.

Appendix 4: Signature Tree

Signature Tree and its Construction: Once we have the skylines found then we can create signature for each skyline. We keep attributes sorted in each skyline. Creating signatures is as follows. We first initialize a bit vector of length *M* (total number of attributes in the database) with default value 0 for a skyline. In our running example, we have *M* = 6 attributes *(AC, Auto Trans, Four Door, Power Brakes, Power Doors, and Turbo)*, so the length of the signature for each skyline would be 6 and the initialize vector of the signature is 000000. Then for each attribute present in the skyline, we set that the corresponding bit in the bit vector to be 1. For example, for skyline s_1 = {*AC, Four Door*}, the signature is *101000*. A signature file contains the signatures of all the skylines (or transaction traditionally). Table 2 shows the signatures of the skylines (signature file) for our running example.

Skyline ID	*Query ID*	*Signature*
s_1	q_1	101000
s_2	q_2	100010
s_3	q_3	001010
s_4	q_4	000110
s_5	q_5	000001
s_6	q_5	010000

Table 2. Signature of Skylines (Signature File)

Definition (Signature tree): A signature tree for a signature file $S = s_1.s_2...s_n$, (where $s_i \neq s_j$ for $i \neq j$ and $|s_k| = d$ for $k = 1, \ldots, n$) is a binary tree T such that

i. For each internal node of T, the left edge below it is always labeled with 0 and the right edge is always labeled with 1.
ii. T has n leaves labeled 1,2, …,n, used as pointers to n different positions of s_1, s_2, …, and s_n in s. Let v be a leaf node. Denote $p(v)$ the pointer to the corresponding signature.
iii. Each internal node v is associated with a number, denoted by $s_k(v)$, denoting which digit will be checked.
iv. Let, $i_1,\ldots,i_h$ be the numbers associated with the nodes in a path from the root to a leaf v labeled i. Then, this leaf node is a pointer to the ith signature in S, i.e., $p(v) = i$. Let $p_1,\ldots,p_h$ be the sequence of labels of edges on this path. Then, $(j_1, p_1)\ldots(j_h, p_h)$ makes up a signature identifier for s_i, $s_i(j_1, \ldots, j_h)$.

Creating Balanced Signature Tree: A balanced signature tree is a signature tree which is completely or almost evenly balanced. The method of building a balanced signature tree is described below. The tree might not be always perfectly balanced, but it would be close to being evenly balanced.

A signature file $S = s_1.s_2 \ . \ . \ . \ s_n$ can be considered as a Boolean matrix. We use $S[i]$ to represent the ith column of S. For our example above, we have the digits of signature represented for the attributes as follows:

Attribute	AC	Auto Trans	Four Door	Power Brakes	Power Doors	Turbo
Digit	1	2	3	4	5	6

We calculate the weight of each $S[i]$, i.e., the number of 1's appearing in $S[i]$, denoted $w(S[i])$. Then, we choose a j such that $|w(S[i]) - n/2|$ is minimum. Here, the tie is resolved arbitrarily. Using this j, we divide S into two groups $g_1 = \{s_{i1}, s_{i2}, \ . \ . \ . \ , s_{ik}\}$ with each $s_{ip}[j] = 0$ ($p = 1, \ . \ . \ . \ , k$) and $g2 = \{s_{ik+1}; s_{ik+2} ; \ . \ . \ . \ ; s_{iN}\}$ with each $s_{iq}[j] = 1$ ($q = k + 1, \ . \ . \ . \ , n$); and generate a tree as shown in Figure 4(a). In fact, we partition the signatures based on the value on column j; signatures with value 0 on column j go into one group and signatures with value 1 on column j go into another group. In a next step, we consider each g_i ($i = 1, 2$) as a single signature file and perform the same operations as above, leading to two trees generated for g_1 and g_2, respectively. Replacing g_1 and g_2 with the corresponding trees, we get another tree as shown in Figure 4(b). We repeat this process until the leaf nodes of a generated tree cannot be divided any more. Considering our running example, we can see that at the first time the sum of 1's in each column $w(S[i])$ is as follows: column 1 (AC) = 2, column 2 = 1, column 3 = 2, column 4 = 1, column 5 = 3, and column 6 = 1. Here, $n = 6$ which is the total number of skylines. So, column 5 has the minimum value for $|w(S[i]) - n/2|$ which is $(3 - 6/2) = 0$. So we choose column 5 which is *Power Doors* as the root of the tree. We follow the same process for each sub-tree from the root. In Figure 4(a), $g_1 = \{s_1, s_5, s_6\}$ and $g_2 = \{s_2, s_3; s_4\}$; and, in Fig. 6(b), $g_{11} = \{s_5, s_6\}$, $g_{12} = \{s_1\}$, $g_{21} = \{s_3, s_4\}$, and $g_{22} = \{s_2\}$. Figure 5 shows the complete signature tree built for the skylines of our running example.

At the leaf node of the tree, we keep information for the skyline as well as the query where it came from. As we recall, our goal is to maximize the number of queries for which the new tuple will be visible on the skylines. We do not want to count skylines from the same query for a candidate set more than once.

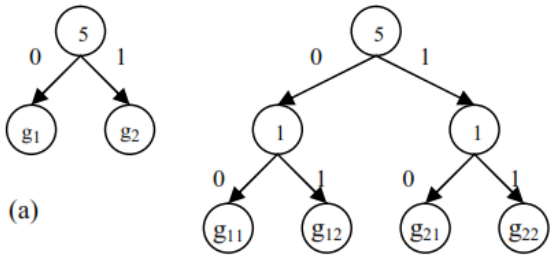

Figure 4. Process of Building Signature Tree

At this step we generate the signature tree only for the skylines with less than or equal to *m* attributes. The reason we ignore the skylines with more than *m* attributes is that none of them can eventually be subset of any *m*-attributes candidate set which we generate in next step. This will be an efficient technique where there are many skylines which have more than *m* attributes present in the skyline log.

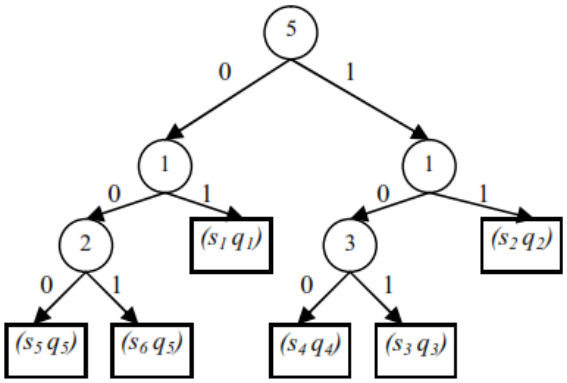

Figure 5. Signature Tree for the Skylines

Appendix 5: Traditional Approach of Searching the Signature Tree

We search the signature tree for the new tuple (*m* attribute set). As in the *Apriori* algorithm (Agrawal & Srikant, 1994), we start with frequent *1*-itemsets (attribute sets). A *minimum support* is used such that when we select top-*m* attributes for the new tuple *t*, then *t* should be on the skylines for the number of queries at least or equal to the *minimum support*. A minimum support is the lower bound such that at least these many queries should have the new tuple visible on their skylines. We use a heuristic method to select a good *minimum support*. We first use a fixed value, for example 1% and execute the algorithm. Then we change the minimum support as required, for example if we find no queries for the new tuple then we decrease the minimum support and if too many queries are found then we increase the minimum support until a good value for minimum support is set. Using the minimum support we generate frequent *1*-itemsets (attribute sets) from the skylines. Here we only consider the attributes which are present in the new tuple to be advertised. One approach now could be to generate all possible *m*-attribute sets using *apriori-gen* function in Figure 3, and then search the tree. We can search the signature tree as follows:

i. Create signature for each of the *m*-attribute candidate sets. Let s_t be the candidate set signature. The *i*th position of s_t is denoted as $s_t[i]$. During the traversal of a signature tree, the inexact matching is done as follows:

   a. Let *v* be the node encountered and $s_t[i]$ be the position to be checked.
   b. If $s_t[i] = 0$, we move to the left child of *v*.
   c. If $s_t[i] = 1$, both the right and left child of *v* will be explored.

   In fact, this process just corresponds to the signature matching criterion, i.e., for a bit position *i* in s_t, if it is set to 0, the corresponding bit position in *s* must be set to 0; if it is set to 1, the corresponding bit position in *s* can be 1 or 0. In a traditional signature tree, a query *q* is passed to the tree and finds the transactions (leaf nodes of the tree) which are possibly the supersets of *q* (i.e., how many transactions will be retrieved by the query). But our problem is different. We pass a candidate *m*-attribute set *c* to the tree and find the skylines (leaf nodes) which are possibly subsets of *c*. Searching of the tree is done in a depth-first manner. When we reach a leaf node, we match all the signatures of the leaf node with *m*-attribute candidate set *c*. Here skylines have to be subsets of *c*. We keep a count for each candidate set *c* that how many skylines have been found as subsets of *c*. Here, we count each query only once. For example, considering our running example, if we find two skylines s_5 and s_6 are the subsets of any candidate set c_i, we only count them once because both come from the same query q_5. As we recall, our problem is not to maximize the number of skylines, but to maximize the number of queries which will have the new tuple on their skylines. We remove the candidate set *c* if the total count for it is less than the *minimum support*.

ii. For all m-attributes candidate sets found in step (i), we simply return the set that has the highest count.

Appendix 6: Major Problems of Traditional Searching of the Signature Tree

There are two major problems with the traditional approach of searching the trees: (a) the number of candidate sets can be huge as there is no pruning at intermediate steps by searching the tree, and (b) small itemsets would get an unfairly small count because it increases the count of a candidate if it satisfies whole skyline itemsets in the signature tree. Hence, in order to be able to grow the candidate itemsets and not start directly from *m*-itemsets, we start generating and searching the tree in order to increase the count of a candidate *k*-itemset for every query it has a chance to cover if (*m*-*k*) items are added. For instance, the *2*-itemset 110000 has a chance to cover 110100 if 1 more item is added. So we follow a new method where for each *k*-itemset we navigate the signature tree from top to bottom and only prune subtrees that need more than (*m*-*k*) additional items to be covered.

Appendix 7: System Configuration and Datasets used for the Experiments

System Configuration: We used Microsoft SQL Server 2000 RDBMS on a P4 3.2-GHZ PC with 1 GB of RAM and 100 GB HDD for our experiments. We implemented all algorithms in C#, and connected to the RDBMS through ADO.

Dataset: We use an *online used-cars dataset* consisting of 15,211 cars for sale in the Dallas area extracted from autos.yahoo.com. There are 32 Boolean attributes such as *AC, Power Locks*, etc. We used a real query log of 185 queries created by university users, as well as synthetic query logs of 1000, and 100000 queries. In the synthetic query logs, each query specifies 1 to 5 attributes chosen randomly distributed as follows: 1 attribute – 20%, 2 attributes – 30%, 3 attributes – 30%, 4 attributes – 10%, 5 attributes – 10%. That is, we assume that most of the users specify two or three attributes.