Efficient Skyline Retrieval with Arbitrary Similarity Measures

Deepak P Prasad M Deshpande Debapriyo Majumdar Raghu Krishnapuram
IBM Research, India Research Lab, Bangalore
{deepak.s.p,prasdesh,debapriyo,kraghura}@in.ibm.com

ABSTRACT

A skyline query returns a set of objects that are not dominated by other objects. An object is said to dominate another if it is closer to the query than the latter on all factors under consideration. In this paper, we consider the case where the similarity measures may be arbitrary and do not necessarily come from a metric space. We first explore middleware algorithms, analyze how skyline retrieval for non-metric spaces can be done on the middleware backend, and lay down a necessary and sufficient stopping condition for middleware-based skyline algorithms. We develop the Balanced Access Algorithm, which is provably more IO-friendly than the state-of-the-art algorithm for skyline query processing on middleware and show that BAA outperforms the latter by orders of magnitude. We also show that without prior knowledge about data distributions, it is unlikely to have a middleware algorithm that is more IO-friendly than BAA. In fact, we empirically show that BAA is very close to the absolute lower bound of IO costs for middleware algorithms. Further, we explore the non-middleware setting and devise an online algorithm for skyline retrieval which uses a recently proposed value space index over non-metric spaces (AL-Tree [10]). The AL-Tree based algorithm is able to prune subspaces and efficiently maintain candidate sets leading to better performance. We compare our algorithms to existing ones which can work with arbitrary similarity measures and show that our approaches are better in terms of computational and disk access costs leading to significantly better response times.

1. INTRODUCTION

The skyline operation is useful in applications that require selecting objects based on multiple criteria. The skyline consists of a set of objects that are not dominated by other objects. Domination is usually assessed with reference to a query object where an object dominates another if it is at least as similar to the query object on all dimensions and strictly more similar in at least one dimension. Top-k retrieval, a related problem, finds objects similar to a given query based on a weighted similarity function where the similarity is, most usually, computed as a monotonic aggregate (e.g., a weighted sum) of similarities in multiple dimensions considered. Skyline query is notably different in that it does not require weighting among the different dimensions. Further, the skyline contains every object that is closest to the query based on any monotone aggregation function of similarities. In addition, for every point in the skyline, there exists a monotone scoring function that is maximized at that point. Thus, the skyline does not contain any object that is not the best according to some possible weighting.

We encountered the skyline problem in the IT service scenario where a system administrator who has come across a problem on a server, seeks to find similar servers since they may have encountered similar problems before so that (s)he could reuse their solutions. In this case, it is difficult to come up with a weighting for attributes as the attributes are as diverse as operating system, network card details, applications and software installed. The skyline of servers is suitable here as it includes all the relevant servers.

1.1 Non-metric Spaces

Many attributes in various applications are categorical and the similarities between the various values often come from domain knowledge e.g., the set of operating systems where a domain expert has defined the similarities for each pair of operating systems. These similarity measures are arbitrary and are most often non-metric. The cardinality of intersection, the most common similarity measure for set-values attributes (such as software installed) is also non-metric. Such attributes are also encountered in a variety of real-world scenarios where skyline queries are of interest. For example, in the case of a hotel search, the set of amenities provided is a commonly considered set valued attribute. While searching for TVs, the display type (Plasma, CRT, LCD) is an important attribute with non-metric similarity measure. It may also be noted that such attributes do not have a total order among their values consistent with the similarity measure. The values can be ordered only with respect to a query based on the distance from the query value. For example, in Figure 1, *Server 1* and *Server 2* constitute the skyline for a query *(DB2, Windows XP)* when the attributes considered are *Database Server* and *Operating System*. In this example, *(Windows Vista, Red Hat Linux, AIX)* are the Operating Systems in the database in the non-decreasing order of dissimilarity from the query value, *Windows XP*. However, there is no absolute order among the values *Windows Vista, Red Hat Linux and AIX* without a

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

query.

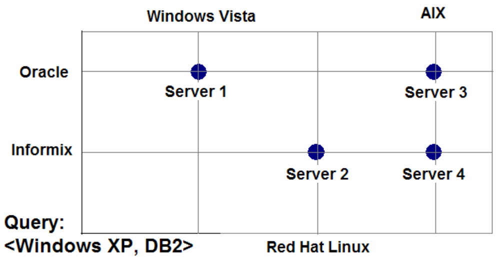

Figure 1: Example

In this paper, we address skyline computation for such arbitrary similarity measures where metric space approaches cannot be used. Points in a metric space satisfy the triangle inequality, i.e., $d(x, y) + d(y, z) \geq d(x, z)$. Most of the spatial indexes such as R-Tree [15, 16, 17], kd-tree [5], and B-Tree [18, 19] use this property to partition the dataset for indexing. However, in a non-metric space, it is not possible to statically partition the space. The value ordering is different for each query and depends on the query values. For example, without the metric property, one cannot characterize each partition with a bounding box as is done in R-tree. Consider two points A and B that are similar and placed together in a R-tree node. Given a query Q, the similarity between A and Q and that between B and Q are independent. A may be very similar to the query Q whereas B might be very dissimilar. One cannot bound the dissimilarity of B based on the similarity of A with Q, as would be required to exploit the index. The only way to use an R-tree would be to create a new index based on the query Q. Recreating the index for each query is infeasible since it involves the scanning of the entire database thus defeating the very purpose of usage of the index. Thus traditional spatial indexes are not suitable for non-metric spaces. A good discussion of metric properties and how they are used in popular indexing structures can be found in [21].

1.2 Related Work

The need for non-metric similarity functions has been argued in [14], which says that the triangle inequality property is too restrictive to model the (dis)similarities as perceived by humans. The skyline operator was analyzed in detail for the first time in [6] and since then the problem of skyline retrieval has been receiving increasing attention. Two flavors of the skyline problem have been studied in literature: (1) Retrieving the skyline for the database [17], and (2) Retrieving the skyline from the database for a given query object. The former problem is applicable only in a database where *all* attributes are from ordered domains. In fact, many of the algorithms for the more general query based skyline retrieval also assume that the data points are from a metric space. Distributed skyline query processing also has been a subject of recent research [20, 9].

Among the algorithms developed previously, the Block-Nested-Loops (BNL) and Divide & Conquer [6] algorithms are the ones which address the problem in the general setting of arbitrary similarity measures. The worst case complexity of BNL is $O(n^2)$ and that of Divide and Conquer is $O(nlog(n) + n(log(n)^{m-2})$ for a dataset of n objects and m attributes. For most practical dataset sizes, Divide and Conquer becomes worse than $O(n^2)$ for $m \geq 5$, thus making it unsuitable for high dimensional datasets. Although many different algorithms and indexing structures have been proposed since [6], most of them are applicable only in restricted settings. For example, the problem has been well studied in ordered attribute domains.

Sort-First-Skyline [7] and LESS [13] employ an initial topological sort of objects to reduce the number of comparisons required. However, such a sorting is impractical in the case of arbitrary similarity measures, as the order is determined based on the dissimilarity with the query (and thus, only at query time).

The skyline retrieval problem is also important in the widely used middleware setting, where we get results from different backend systems and then compute a skyline over these results. This setting has been widely studied in the context of top-k retrieval (by [11],[12] and many others). Adapting the middleware algorithms for top-k to compute skylines efficiently is not trivial since the nature of the domination relation in case of skyline is quite different from the score aggregation based top-k retrieval. Middleware algorithms for computing skyline have been first explored by Balke et. al [3]. They primarily focus on minimizing the fraction of the lists visited by the algorithm and do not analyze the total access cost which is a major bottleneck in case of middleware algorithms.

1.3 Our Contributions

In this paper, we propose algorithms that work in the very general setting where similarity measures can be arbitrary. In the first part we consider the middleware setting, where we develop an algorithm, namely the *Balanced Access Algorithm (BAA)*, which is provably and experimentally better than the one in [3] in terms of IO costs. We analyze the skyline problem in the middleware setting and present several insights into nature of the problem. Our analysis shows that BAA is the best-effort middleware algorithm without prior knowledge about data distributions thus indicating that it is not possible to improve significantly over BAA in the middleware setting. *Further, we empirically show that BAA is very close to the absolute lower bound of IO costs for middleware skyline algorithms.*

In the second part, we explore a non-middleware setting where we have more control over the backend so that we can build suitable indexes to optimize skyline retrieval. Recently, the Attribute Level Tree (AL-Tree) [10] has been proposed as a suitable indexing structure for top-k computation with arbitrary similarity measures. In this paper, we develop an algorithm that uses the AL-Tree index to compute the skyline efficiently and experimentally show that it outperforms the middleware algorithms by a huge margin.

Our main contributions can be summarized as:

- A middleware algorithm for computing the skyline and their evaluation
- Analysis of computing skyline for any middleware algorithm
- An empirical comparison between our BAA algorithm and the absolute lower bound of IO costs for middleware skyline algorithms which shows that BAA is reasonably close to the absolute lower bound

- An AL-Tree based algorithm for computing the skyline that can be used in a general setting
- A detailed experimental evaluation of our algorithms along with the existing algorithms (namely, Balke's middleware algorithm and the BNL algorithm) which show that our algorithms outperform the previous works in all cases.

In Section 2, we describe the problem formally. We develop the skyline algorithm for middleware in Section 3 and the AL Tree based algorithm in Section 4. We present our experimental evaluation in Section 5 and finally conclude in Section 6.

2. PROBLEM DEFINITION

We will now define the skyline query problem formally. Let D be the set of objects in the database. Assume that each object in D has m attributes each. The dissimilarity function d_i for attribute i is a function $d_i : A_i \times A_i \rightarrow \Re$ where A_i is the domain of attribute i. An object X is said to dominate another object Y with respect to a query object Q, (represented as $X \succ_Q Y$) if X is at most as dissimilar from the query on *each* attribute as Y and there exists at least one attribute on which X is more similar to Q than Y:

$X \succ_Q Y$ iff

1. $\forall i, d_i(v_i(X), v_i(Q)) \leq d_i(v_i(Y), v_i(Q))$ *and*
2. $\exists i, d_i(v_i(X), v_i(Q)) < d_i(v_i(Y), v_i(Q))$

where $v_i(O)$ is the i^{th} attribute of object O. We use $d_i(X, Q)$ as a shorthand for $d_i(v_i(X), v_i(Q))$ in subsequent sections. It may be noted that the second condition above ensures that duplicates (i.e., objects which have the same value for all attributes) do not dominate one another.

The skyline query problem is defined as follows:

Definition 1. **Skyline Query Problem:** Given a query Q, find *all* objects from D that are not dominated (with respect to Q) by *any* other object in D. This corresponds to finding the set $S \subseteq D$, such that the following conditions are satisfied:

1. $\forall s \in S, \nexists d \in D$ such that $d \succ_Q s$ *and*
2. $\forall t \in (D - S), \exists s \in S$ such that $s \succ_Q t$

The first condition means that there are no objects dominating the ones in S whereas the second condition ensures that S covers all non-dominated objects. In certain cases where the user may not want to distinguish between duplicate objects, S may be pruned to remove the duplicates. As mentioned in Section 1, we consider the case where $d_i(.,.)$ is not a metric. It may be observed that $S = D$ in the worst case, but, in most practical scenarios, $|S| << |D|$. Further, it may be noted that cases were S contains most of the objects in D may not be interesting to the user as (s)he would then have too many results to analyze.

3. SKYLINE ALGORITHMS FOR MIDDLEWARE

Middleware algorithms work by accessing sub-systems to fetch <object-id, dissimilarity score> pairs, and by combining them efficiently to arrive at a result set. Typically, there are as many sub-systems as attributes, each sub-system provides objects (usually, identifiers for objects) along with their dissimilarity scores in the *non-decreasing order of dissimilarity from the query* on that attribute, when probed sequentially. The middleware algorithm does not have control over the subsystems that it accesses. However, even in cases of non-metric spaces, each subsystem could be implemented in a way that does not need to do sorting of the entire database according to that query using indirection lists (according to the attribute that the subsystem represents); we omit those details since we focus on middleware algorithms in this section. Sub-systems also allow random access to retrieve the similarity score of an object to the query. Depending on the system and implementation, random accesses are typically $10 - 50,000$ times costlier than sequential accesses, so the algorithm needs to carefully balance between these two types of accesses to optimize the access cost. In this section, we propose a *Minimum Stopping Condition (MSC)* and show that any middleware algorithm for skyline retrieval can claim to be correct **if and only if** it satisfies MSC. Further, we propose a middleware algorithm for skyline retrieval, the *Balanced Access Algorithm (BAA)* that seeks to balance random and sequential accesses after MSC in a bid to optimize on IO costs. We prove that BAA outperforms the algorithm proposed in [3] on *every* dataset. We analyze how the uncertainties about domination relationships between objects get resolved incrementally as any middleware algorithm sees more and more objects, and show that BAA is a very reasonable best-effort strategy for general settings.

3.1 Minimal Stopping Condition

Theorem 1. A middleware algorithm that does not make *wild guesses*[1] can claim that it has seen all distinct objects[2] in the skyline (at least in one list) ***if and only if*** it has seen at least one object O such that:

$$\forall_i, d_i(O, Q) \leq max\_diss\_so\_far_i$$

where $max\_diss\_so\_far_i$ is the maximum dissimilarity score seen on the i^{th} list through sequential accesses (objects with the same dissimilarity score on any list may be stored in any arbitrary order in the list). We refer to this condition as the *Minimal Stopping Condition (MSC)*.

Proof. We first show that attainment of MSC is a sufficient condition for stopping. At MSC, let O' be a so-far-unseen object. By virtue of the ordering of the lists, the following holds:

$$\forall_{i=1}^{m}, d_i(Q, O) \leq d_i(Q, O')$$

Now, O' could either be a dissimilarity-wise duplicate of O or may be dominated by O if it is farther away from O' on at least one attribute. Either way, it cannot be a distinct object in the skyline. Now, assume that an algorithm claims to have seen every distinct object in the skyline before reaching the MSC condition. Since the algorithm cannot assume

[1] An algorithm that makes wild guesses may see an object through random access before it sees it through sequential access [12].

[2] Two objects are considered non-distinct if they are dissimilarity-wise duplicates, i.e., take the same dissimilarity score on all attributes. We refer to dissimilarity-wise duplicates, as simply duplicates in later sections.

anything about the unseen objects, let there exist an object O' such that:

$$\forall_{i=1}^{m}, d_i(Q, O') = max\_diss\_so\_far_i$$

and that O' is the next item to be seen by the algorithm through sequential access on all the lists. Now, we argue that O' is in the skyline. As MSC has not been reached yet, for every object O so far seen, there exists at least one attribute where its dissimilarity from the query is strictly greater than $max\_diss\_so\_far_i$, i.e., $\exists_i d_i(Q, O) > max\_diss\_so\_far_i$;$O'$ is closer to the query than O on that attribute. It follows that O' is in the skyline, negating the algorithm's claim. □

If the scores on every list strictly increase (and not remain constant), the MSC condition simply means that the algorithm must see one object in all the lists through sequential accesses. In presence of such duplicates, attainment of this condition does not imply that we have seen all skyline objects, as duplicates of objects which enabled attainment of the condition may not have been seen yet. We will see in Section 5 that MSC is usually achieved very early. However, in certain cases, the algorithm may have to read significantly deep into the lists to achieve MSC. MSC is a generalization of the condition in [3] and outlines the minimal stopping condition for *any* middleware algorithm, as opposed to the latter which is an *if and only if* condition for only those algorithms which do only sequential accesses. MSC is the earliest possible point at which all distinct objects in skyline have been seen at least once by an algorithm which does both sequential and random accesses. The condition in [3] is less effective as it always occurs after the MSC.

3.2 Balanced Access Algorithm (BAA)

The MSC condition ensures that the algorithm has seen all objects in skyline at least once. Besides reaching the MSC, the algorithm has to ensure that all uncertainties regarding domination relationships between objects are resolved to ensure that the skyline objects are correctly identified, from among the objects seen until MSC. In this section, we propose the Balanced Access algorithm, which can be summarized to have the following properties:

1. Does Round Robin sequential accesses until MSC.
2. Uses estimates of random access and worst case sorted access costs to schedule accesses after the MSC.

At any time, BAA maintains the set of fully seen and so-far non-dominated objects in F, and the set of partially seen objects in P. At any point, it does random accesses if the candidate set cannot be held in memory. This ensures that BAA is able to work with enough memory to hold $O(|S|+\theta)$ candidates, where S is the set of Skyline objects (the output) and θ is the length of the longest sequence of entries with the same dissimilarity score. Once MSC is reached, it estimates whether doing random accesses is profitable by comparing the estimated cost of *scanning the remainder of the lists sequentially* and *the cost of doing random access on all candidates in P*. Once past MSC, it heeds only those objects which are already under consideration (Ref. Theorem 1 and Line 10 in Algorithm 1). BAA is able to utilize any extra memory available to delay random accesses till MSC, and thus may avoid some (on objects which BAA sees soon through sequential accesses).

Alg. 1 ***Balanced Access Algorithm (BAA)***

```
1   $F = \phi$ /*the set of fully seen objects*/
2   $P = \phi$ /*the set of partially seen objects*/
3   while($\neg$ MSC $\vee$ P $\neq \phi$)
4     if(out of memory $\vee$ (MSC $\wedge$ random profitable))
5         do random access on all objects in $P$
6         perform checks on $F \cup P$ and update
7     else
8         do one sequential access on each list
9         Let $P'$ = bag of objects seen in this iteration
10        if(MSC)   $P' = P' \cap (F \cup P)$
11        Update the sets $F$ and $P$ based on $P'$
12        if(any object newly became fully seen)
13              perform checks on $F \cup P$ and update
14        If $MSC$ is reached, set the flag
```

The algorithm proposed in [3] does round robin accesses until one object is seen fully on each of the lists. It then does sorted accesses on each of the lists separately until a score change, to ensure that duplicates of the fully seen object are not missed. At this point, all candidates are accessed randomly and skyline objects are identified. To ensure level ground for the comparison, we adopt the variant of the algorithm which starts random access when one object becomes fully seen, making the algorithm duplicate insensitive. This adaptation is advantageous to the algorithm in that lesser number of candidates are seen, thus reducing the IO costs. We refer to this variant as simply Balke, in the rest of the paper.

Lemma 1. Provided with the same amount of memory, Balke is *never* better than BAA in terms of IO costs.

PROOF. Due to BAA having as much memory as Balke, it never has to do random accesses until MSC; this delays the detection of attainment of MSC till the point when at least one object is seen through sequential accesses on each of the lists, which is when Balke starts to do random accesses on all its candidates. Let the number of candidates at this point be c. The cost incurred by Balke from hereon is c random accesses i.e., $(c * c_r)$ (where c_r is the cost of one random access). Let BAA's worst case sequential access cost estimate be s'. If $(c * c_r) < s'$, BAA does random accesses on all candidates, incurring as much cost as Balke. If $(c * c_r) > s'$, it starts doing sequential accesses, and may switch to random access if that becomes more profitable. In either case, the cost incurred is bounded by s', which is in turn lesser than the cost incurred by Balke. □

3.3 Analysis of Skyline Algorithms

In this subsection, we analyze the properties of middleware skyline algorithms, and show that BAA is designed to exploit them. At MSC, every skyline algorithm would have seen a superset of skyline objects. The task, from hereon, is to identify the subset of skyline objects from those seen so far. Let $X \rhd_W Y$ denote the information that $\forall_{i \in W}, d_i(X, Q) \leq d_i(Y, Q)$ and that the relationship between X and Y for other attributes in $\{1, 2, \ldots, m\} - W$ are unknown. Such a relation may be represented as a directed graph (viz., *the dependency graph*), edges representing the relation between candidates (nodes). We evolve the relation

$X \rhd_W Y$ using two operations (as more and more data is seen):

- If Y is not closer to Q on any attributes in $\{1, 2, \ldots, m\} - W$, the edge and the node Y are dropped.
- If Y is closer to Q on at least one attribute in $\{1, 2, \ldots, m\} - W$, the edge is dropped.

This graph has the property that any object with no edges pointing to it would be part of the skyline. It is easy to prove that the status of objects with inward edges can be confirmed only after the uncertainty with respect to the edge is removed. Every skyline algorithm would fit into an abstract framework in Algorithm 2. The different algorithms may differ in terms of the strategies adopted in Step 1 and Step 3 and in using weaker conditions in Step 1 and Step 2. It may be noted that algorithms may not explicitly maintain the dependency graph. The following are notable properties (we omit proofs because of space constraints):

Property 1. Let $\rhd$ be a relation depicting the existence of any edge $\rhd_W$ (for any W). This relation is transitive.

Property 2. For every algorithm that interleaves random and sequential accesses after MSC, there exists an algorithm which has a pure sequential access phase followed by a pure random access phase and incurs IO costs *at most* as much as the former. In other words, performing random accesses in between sequential accesses does not help in reducing IO cost.

Property 3. At any point after MSC, let the number of nodes with at least a degree of 1 be p. Any algorithm that decides to do only random accesses from thereon, cannot claim to be correct unless it performs at least p random accesses. This follows from Property 1 above.

The *Balanced Access Algorithm* is designed to exploit the above properties. Due to the absence of any information about attribute dependencies, we schedule sequential accesses in a round-robin fashion. Guided by Property 2 above, BAA delays random accesses till the end. According to Property 3, any algorithm can maintain an exact estimate of the cost of a pure random access phase from thereon. *Thus, algorithms which reach the same MSC can differ in costs only by means of varying the amount of sequential accesses performed.* In the course of our experiments, we observed that, by not doing any random accesses, an algorithm has to go very close to the end of the lists to be able to identify skyline objects. BAA, because of being ignorant about object distributions in remainder of the lists, maintains an estimate of accessing the remaining parts of the lists fully (which is very close to the cost of a pure sequential access algorithm, given the empirical observation), compares it with the random access cost estimate and switches to the random access phase at such a point when the random access cost estimate is profitable. Intuitively, BAA is close to the optimal middleware algorithm for skyline retrieval. In Section 5.3.3 we show that the IO cost incurred by BAA is actually very close to an experimentally computed lower bound of the IO cost for any middleware algorithm. Thus, the IO cost for middleware algorithm cannot be minimized much further than what BAA does.

However, when we have more control over the storage, we can potentially do better, as we will see in the next section.

Alg. 2 ***Abstract Skyline Algorithm***

```
1   Access the lists until MSC is attained
2   while(dependency graph has edges)
3       Perform more access(es) on the lists
4       Evolve the graph using new info
5   Output all nodes
```

Id	OS Name	Memory
1	MS Windows (MSW)	512M
2	MS Windows (MSW)	2048M
3	RedHat Linux (RHL)	2048M
4	SuSE Linux (SL)	1024M
5	SuSE Linux (SL)	1024M

Table 1: Sample dataset

4. SKYLINE RETRIEVAL USING AL-TREE

In this section, we briefly describe the AL-Tree (a value space indexing structure proposed in [10]), and propose algorithms that use the AL-Tree for efficient retrieval of skylines. Further, we describe heuristics that can speed up the algorithms by pruning parts of the tree structure, thus optimizing on IO and computational costs.

4.1 The Attribute Level (AL) Tree

Consider the database D and a specific ordering of attributes. Each database object can now be represented as a sequence of values, the i^{th} value in the sequence corresponding to the i^{th} value in the chosen attribute ordering. The AL-Tree for D using the chosen ordering is then precisely the prefix tree[3] for the ordered database. In such a tree, all the leaf nodes would be at the same level, i.e., level m, (as every object has the same number of attributes) and each level in the tree would correspond to a specific attribute, according to the chosen ordering. The tree is compressed by collapsing each chain in the tree to the head of the chain; such compressed chains form leaf nodes at levels lesser than m. Each leaf in the tree maintains information about the objects that it stands for, and also any values for remaining attributes (in cases of leaf nodes representing collapsed chains). Any object in the database is uniquely associated with a leaf node in the AL-Tree, and all duplicate objects map to the same leaf node in the tree. For any node N in the AL-Tree, we use $Lvl(N)$, $Obj(N)$ and $Val(N)$ to denote the level, the set of all descendant objects, and the value corresponding to the node. We use the relation $X \rhd Y$ to denote that X occurs before Y in the depth first traversal of the ordered tree.

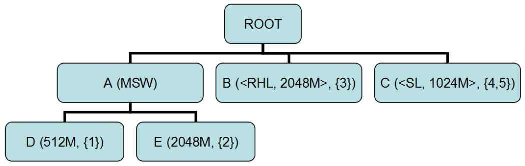

Figure 2: A Compressed AL Tree.

[3] http://en.wikipedia.org/wiki/Trie

Example 1. Consider the dataset in Table 1, and the corresponding AL Tree. Note that for OS Name of value 'RHL', there is a single value of Memory ('2048M') in the database. So the chain can be combined into a single node B represented by $(< RHL, 2048M >, \{3\})$. Similarly, the chain with OS Name of value 'SL' and Memory of value '1024M' can be compressed into node C. The compressed tree with both the chains compressed would be as in Figure 2.

4.2 Handling Non-metric Spaces using Indirection Lists

In the sample AL-Tree in Figure 2, we assumed a certain ordering of values for each attribute. However, in the case of non-metric spaces, we have seen that the attribute ordering is query specific and not absolute. During query processing, we would like to process siblings in the non-decreasing order of dissimilarity from the query (based on the attribute corresponding to the level of the siblings). Constructing a different AL-Tree for each query is clearly infeasible since it involves at least one scan of the database. We accomplish query time sibling ordering in the AL Tree using precomputed indirection lists, which we describe in this section. Let the values that attribute a_j can assume be denoted by be $A_j = \{A_{j_1}, A_{j_2}, \ldots A_{j_{c_j}}\}$. For every value v of attribute a_j, we maintain a list L, of values from A_j in the non-decreasing order of distance (non-increasing order of similarity) from v, i.e. the following holds,

$$d_j(L[p], v) \leq d_j(L[q], v), \forall p < q$$

For every attribute a_j, there would be c_j such lists (one list per value from A_j), each of length c_j. The total size of the collection of lists would hence be $\Sigma_j ({c_j}^2)$ values. *However, as these lists are held on disk and each query needs to get only as many lists as the number of attributes (the ordering for children of sibling nodes would be the same for the same query as the similarity lists are per attribute-value entities), this approach is scalable.* These similarity lists specify the ordering for siblings of internal nodes. Each internal node N, in the AL Tree would have a value based lookup function for the children nodes which can be defined as follows:

$$Child_N(v) = \begin{cases} C, \text{if } \exists C, \ni: (PARENT(C) = N) \wedge (V(C) = v) \\ null, otherwise. \end{cases}$$

Both the tree and the indirection lists are stored in disk. At query time, the indirection lists corresponding to the query values are retrieved upfront from disk so that the traversal algorithm can consult them to ensure that it traverses siblings in the desired order. More details of the implementation can be found in [10].

Example 2. Consider the dataset in Table 1, and the query $(RHL, 2048M)$. The list for the value RHL (i.e., values of the same attribute in the decreasing order of similarity) would thus be $< RHL, SL, MSW >$ (the second row in the OS Similarity Matrix) and that for the value $2048M$ would be $< 2048M, 1024M, 512M >$. The re-ordered AL Tree for the query $(RHL, 2048M)$ is shown in Figure 3. Consider the scenario in the search where we have to find the 2^{nd} child of A. The function progresses through the list $< 2048M, 1024M, 512M >$, firing value based lookup queries for each of the values in the list until it finds the 2^{nd} non-null child and returns it. The sequence of queries would be $Child_A(2048M)$, $Child_A(1024M)$ and $Child_A(512M)$. The iteration stops after it makes the call $Child_A(512M)$ because it finds the 2^{nd} non-null child then.

OS

MSW	MSW, RHL, SL
RHL	RHL, SL, MSW
SL	SL, RHL, MSW

Memory

512M	512M, 1024M, 2048M
1024M	1024M, 512M, 2048M
2048M	2048M, 1024M, 512M

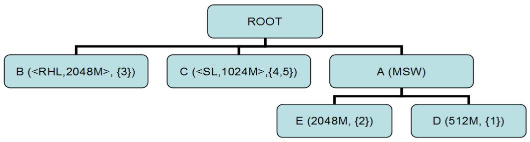

Figure 3: Similarity Lists & a Re-Ordered AL Tree.

The indirection enables us to address another issue. For a certain attribute, there may be multiple values equally dissimilar to the query value of that attribute; in such cases, we virtually merge the corresponding nodes on the tree and treat them as a single node, for the purpose of traversing the tree for the specific query. Such merging can be done trivially by using the indirection lists at tree traversal time; we omit the details here. In particular, for a query value 1280 for the second attribute in the AL-Tree in Figure 2, the nodes D and E may be equidistant (as they have the values 512 and 2048 for that attribute respectively). They would be treated as a single node while traversing the tree for such a query.

4.3 Searching the AL-Tree for Skyline

Certain properties of the AL-Tree make it very suitable for skyline query processing. We discuss such properties in this section, and propose an algorithm which uses them to efficiently retrieve skylines. In the rest of the paper, wherever there is no ambiguity, for sake of simplicity, we refer to the AL-Tree ordered according to a specific query Q and with any mergers necessitated by the discussion in Section 4.2 already done. Some AL-Tree specific notations used in the rest of the paper are summarized in Table 2.

$Obj(N)$	Union of objects corresponding to all descendants of N
$X \triangleright Y$	*true* if X occurs before Y in DFS traversal of the tree ordered according to the query, *false* otherwise
$RightSiblings(X)$	the set of all Y which are siblings of X and $X \triangleright Y$
$min_i(F, g)$	denotes the value in F which is least distant to the value g according to the dissimilarity function $d_i(.,.)$

Table 2: Notations

Lemma 2. For any pair of nodes in the AL-Tree $< X, Y >$, if $X \triangleright Y$, no object in $Obj(X)$ can be dominated by *any* object in $Obj(Y)$:

$$X \triangleright Y \Leftrightarrow \forall x \in Obj(X), \nexists y \in Obj(Y) \ni y \succ_Q x$$

Proof. As $X \triangleright Y$ is *true*, there exists at least one attribute, the one at the level of the deepest common ancestor, where each object of $Obj(X)$ takes a value closer to the query than each object of $Obj(Y)$. Thus, no object in $Obj(X)$ can be dominated any object in $Obj(Y)$. □

Lemma 2 is true for every ordered case; however, the uniqueness in this context is that the AL-Tree enables the algorithm to traverse objects in an ordered fashion for *every* query. This property leads to a simple algorithm which traverses the leaf nodes of the tree in the DFS order, maintaining a set of so-far non-dominated set of objects at any point. Such an algorithm would have to see the entire tree, and would be forced to make a lot of random accesses (no packing of the tree on disk is likely to ensure full sequential accesses, as the order of traversal of the nodes is decided at query-time), thus making it prohibitively expensive in terms of disk access costs. Pruning sections of the AL-Tree can go a long way in reducing the disk access cost. We now enumerate various properties of the AL-Tree and present an improved algorithm which can prune out parts of the tree. *We focus on optimizations which need only a constant amount of additional memory, being fully aware that an unbounded buffer may enable a lot more optimizations.* The heuristics do not improve the complexity of the algorithm; we empirically show that they improve the performance of the algorithm quite drastically(Section 5).

Lemma 3. For any internal node X at level h, if there exists an object x in $Obj(X)$ that takes the closest possible value to the query for every remaining attribute, no sibling Y of X where $X \triangleright Y$ can have a object in the Skyline.

$$\exists x \in Obj(X), \forall_{i=h+1}^{m} v_i(x) = min_i(A_i, v_i(Q)) \Rightarrow$$

$$\forall Y \in RightSiblings(X), Obj(Y) \cap S = \phi$$

Proof. We give an informal argument as proof. For every sibling Y of X, every object in $Obj(Y)$ would take the same values for the first h attributes. If $X \triangleright Y$, any object in $Obj(Y)$ would take a value for the $h+1^{th}$ attribute which is farther away from the query than that of x (since internal nodes which are equally close to the query have already been merged). For the remaining attributes, it can come only as close to the query as x since x already takes the closest values to the query. Thus, $x \succ_Q y$ holds for any y in $Obj(Y)$ eliminating y from S. □

Lemma 4. For every internal node X at level $m-1$, only the first child may lead to objects in the skyline.

Proof. All children of X take the same value for the first $m-1$ attributes. The objects in the first child dominate all objects in other siblings as they are closer to the query in the only attribute in which they differ (i.e., the m^{th} attribute). □

The simple DFS algorithm involves comparing every leaf node to each object in the partial skyline so far (to check for domination). At any internal node, there is a possibility that no such check would succeed for any of its descendant leaf nodes. To avoid such wasteful checks, it would help to have a light-weight function which flags nodes as $not-useful$ upon being able to identify that none of its descendants would succeed the check. We model such a strategy by employing a *function f(X) for every internal node X which returns a set of objects such that the partial skyline so far not being able to dominate all of them is a necessary pre-requisite to be able to find at least one skyline object among the descendants of the node in question:*

$$(\forall x \in f(X), \exists s \in S', s \succ_Q x) \Rightarrow Obj(X) \cap S = \phi ..(1)$$

A function which returns the skyline among the descendants of X is such a function, but its output is bounded only by $|Obj(X)|$. Regardless of the size of the overall skyline of the dataset, any subset of the dataset may have a skyline of the size of the subset. Since we have want to employ only constant memory, we give a linear function for $f(X)$, $\beta(X)$ which returns a single virtual object γ per internal node:

$$\forall_{i=Lvl(X)+1}^{m} (v_i(\gamma) = min_i(\{v_i(x) | x \in X\}, v_i(Q)))$$

i.e., for each attribute under X, γ takes the closest value of that attribute which exists in the subtree rooted at X. The values for the initial $Lvl(X)$ attributes of γ are fixed according to the choices made to reach the node X. An object γ takes may not necessarily exist in the database. The function $\beta(.)$ has a complexity of $O(|Obj(X)|)$. By design, γ dominates all objects in $Obj(X)$ (except duplicates of γ, if they exist). Thus, the partial skyline so far not dominating $\beta(X)$ is a *necessary, but not sufficient* condition for any object in $Obj(X)$ to be in the skyline (domination is transitive). We employ some node pruning in $\beta(.)$ on the lines of Lemma 3. *The saving in terms of computational expenses is that of not having to compare every leaf node under $non-interesting$ (as assessed by β) nodes with the partial skyline.* We present the algorithm as Algorithm 3.

Alg. 3 *SkylineDFS*

```
Input: Node $X$, Bag $S'$
 1    $if(X.isLeaf())$
 2        $S'' = \{x | x \in Obj(x) \& (\nexists s \in S', s \succ_Q x)\}$
 3        Output $S''$
 4        Return $S' \cup S''$
 5    $if(Lvl(X) = (m-1))$ (Ref. Lemma 4)
 6        $return\ SkylineDFS'(X.firstChild(), S')$
 7    $\forall c \in X.Children()$
 8        $if(S' \succ_Q \beta(c)) continue;$(Ref. Equation. 1)
 9        $S'' = SkylineDFS'(c, S')$
10        $if(closest(c) \in (S'' - S'))$ (Ref. Lemma 3)
11            $return\ S''$
12        $S' = S''$
13    return $S'$
```

4.4 Analysis

The *SkylineDFS* algorithm has a worst case complexity of $O(m|S||D|)$. In step 2 (Ref. Algorithm 3), the algorithm checks for objects in S' that can dominate objects in $Obj(X)$ when it reaches a leaf node, X. Lemma 2 ensures that $S' \subseteq S$. As the check needs to be be performed for every leaf node accessed, the complexity of the algorithm is $O(m|S||D|)$.

SkylineDFS is an ***online*** algorithm. It outputs each object in the skyline (Step 3) as and when it processes the leaf node associated with the object (Ref. Lemma 2). ***It is***

very interesting to note that the first leaf node that the algorithm processes would have all its objects in the skyline and it is reached within m steps of the algorithm. The first leaf node is reached by following the first child of the root, its first child and so on. As the tree has a height of at most m, the first leaf node is reached within m such steps.

The variable memory requirement of the basic SkylineDFS algorithm is bounded by the size of the output, i.e., $|S|$. The algorithm maintains S', the bag of skyline points seen so far, which would eventually grow to reach S. Although certain objects may be identified as skyline points much early in the process, such points have to be maintained in S' to eliminate any objects (yet to be seen) which may be dominated by them. The height of the tree, being bounded by m, ensures that the $SkylineDFS$ would have to maintain at most m frames in the stack.

4.5 Comparison with Sort Based Skyline Algorithms

SkylineDFS is similar in spirit to other sort based skyline algorithms developed previously [7, 13] that sort the data before computing the skyline. We point out the major differences here:

- As discussed earlier, in the case of a non-metric dataset, there is no unique ordering and the data would have to be re-sorted for each query. This is very expensive since it involves at least a scan of the entire dataset. In the AL-Tree based algorithm, the tree is computed only once. The tree is traversed in a query specific order using indirection lists, thus saving the cost of building the index per query.

- Other sort based algorithms use some pruning to avoid reading in the entire sorted list. However, the pruning occurs only at the end of the list since the list is basically a linear structure. On the other hand, the AL-Tree based algorithm prunes both leaf level and internal nodes, leading to better pruning.

5. EXPERIMENTS

In this section, we describe our experimental results. We perform a detailed study of the proposed middleware algorithms, comparing them based on various performance measures, most of which are specific to the middleware setting. We then do an in-depth analysis of the proposed algorithms against the BNL algorithm, the state-of-the-art skyline algorithm which handles arbitrary similarity measures.

5.1 Experimental Setup

We compare the various algorithms based on multiple performance measures. Most prominent among them is the response time, which is parameterized by the computational and disk access costs. Our experiments were conducted on an IBM X Series machine with Windows Server 2003 on an Intel Pentium 3.4 GHz Processor and 2.0 GB of RAM.

IO costs are measured in terms of sequential and random IOs. Studies on middleware algorithms have assumed the ratio of random item accesses, c_r to to the cost of sequential item accesses c_s to be between 10 and 50000. Although an item is the conceptual unit of access, disk accesses are typically done at the page level. If a page can hold t items, the ratio of costs between random page accesses, p_r and sequential page accesses, p_s would be $(c_r/c_s)/t$. We assume a t of 1000, thus setting (p_r/p_s) to 10 and $(c_r/c_s) = 10000$, consistently in our experiments. The aggregate IO cost is computed as the ratio-based weighted sum of the IO costs.

Middleware algorithms mostly do sequential accesses (which usually corresponds to the disk packing order) and thus never revisit pages, making them insensitive to increasing the cache size beyond one page per attribute. However, the ALT algorithm can make use of additional cache as it sees nodes in an order different from the disk packing order and may revisit nodes. Unless otherwise mentioned, we use a LRU cache of the size of 7.5% of the dataset size for our experiments. It is particularly advantageous for middleware algorithms to have as many disks (or diskheads) as there are attributes, so that m pages (one per attribute) are accessible by sequential IO at any given configuration. Such a setting necessitates fine-grained control over the storage and makes the backend configuration dataset dependent; hence, we employ only a single diskhead for data access when comparing middleware algorithms with others. It may be noted that such a setting, besides being more realistic, ensures fairness in comparison, as the ALT and BNL algorithms do not require multiple diskheads. When a database of objects stored in sorted order (sorted according to a unique id) is available(or some additional information [4]), middleware random access translates to a single random *page* access. For quantifying page IO costs of middleware algorithms, we assume such a case, and sort objects before random accesses to optimize IO costs. For the disk-based implementation of the ALT algorithm, we do breadth first packing of the AL-Tree.

To isolate the computational costs from the IO costs, we use a scenario where all the objects and indexes are loaded in memory when all costs become purely computational (as IO is eliminated). The metric which is usually considered to be of high significance (being the only measure that is visible to the user) is *response time* for a disk based implementation. For our experiments, we simulate the disk based implementation where we assume page access costs to be 1 ms and 10 ms for sequential and random access respectively using a page size of 40 KB. These estimates are in tune with reported figures on popular platforms [1] [8].

5.2 Datasets

Since we are interested in analyzing the performance of the various algorithms in a very general setting, we use synthetic datasets upfront to illustrate their behavior by varying the data density. Data density is computed as the ratio of the number of data objects to the total possible number of distinct tuples in the space. We generate synthetic datasets with uniform random distribution and random dissimilarities between different values of each attribute. We also run the experiments two real datasets,ForestCover[4] and Census-Income[5], from the UCI Machine Learning Repository [2]. The *ForestCover* dataset contains data of the Forest Cover type for 581012 cells, each of size 30×30 meters over regions in the US. The attributes chosen from the dataset had 67, 551, 2, 700, 2, 7 and 2 distinct values (The dataset has as many as 44 binary attributes among the 55 total attributes present) leading to a data density of 0.04%.

[4] http://kdd.ics.uci.edu/databases/covertype/covertype.data.html

[5] http://kdd.ics.uci.edu/databases/census-income/census-income.html

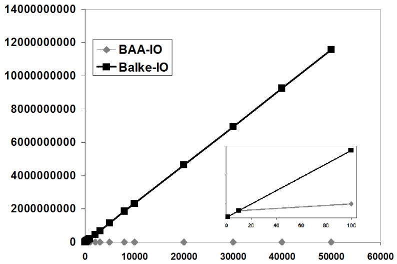

Figure 4: IO-Cost vs Varying Ratio

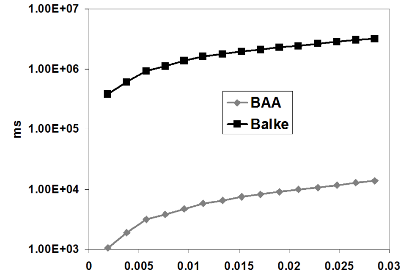

Figure 5: Response Time vs Density (Varying Data Size)

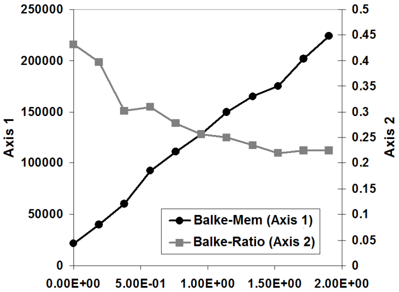

Figure 6: Balke Stats vs Density (Varying Data Size)

The $Census - Income$ Dataset contains census data for 199523 people for 1970, 1980 and 1990 from the Los Angeles area. We choose a subset of attributes, namely *Age, Education, Number of Minor Family Members, Number of Weeks Worked and Number of Employees*, from the dataset, based on their utility in measuring similarities between people. The attributes chosen have 91, 17, 5, 53 and 7 distinct values respectively leading to a high density of 6.9%.

5.3 Middleware Algorithms

In this section, we compare the BAA and Balke algorithms in a *pure* middleware setting where IO costs are measured as the weighted sum of sequential and random item accesses. We do not have any control over the sub-systems in the middleware scene, thus making page-access estimation impossible. Based on the implementations of the sub-systems, the c_r/c_s ratio may vary widely. For example, a sub-system which maintains in-memory lists may have a ratio close to 1, whereas a disk-based implementation may have a much higher ratio. We have shown in Lemma 1 that BAA is always better than Balke in terms of IO costs. For all experiments in this section, we assume that both the algorithms are provided with the same amount of memory.

5.3.1 Performance Comparisons

We analyze the performance by varying the c_r/c_s ratio from *1* to *50000* on a million object dataset with *5* attributes. The Balke IO cost was observed (Figure 4) to increase linearly with the ratio, primarily because of the large number of random accesses (whose cost is linear on the ratio) whereas BAA has very low IO costs. BAA switches between random and sequential accesses, and does lesser and lesser number of random accesses with increasing ratios. On very high ratios, BAA evolves into a no-random-access algorithm. The contrasting rates of increase is more visible at lower ratios (inset in Figure 4).

In a different set of experiments, we vary the data density by varying the dataset size from 100,000 to one and a half million, in the process varying the density from 0.09% to 2.8%. We fix the c_r/c_s ratio at 10000 and assume that each sequential access takes 1 μs and each random access takes $10ms$. These figures are derived from literature [1] [8] assuming that each page can hold around 1000 items. This quantification of IO costs in terms of time enables us to aggregate IO and computational costs on a time scale, and report response times. BAA and Balke both deteriorate with dataset size and have a similar rate of deterioration (Figure 5) with BAA consistently outperforming Balke by more than two orders of magnitude. BAA has a reponse time of around 10 seconds even on a dataset of size one and half million whereas Balke takes close to half an hour on the same. In a separate set of experiments, we vary the data density from 0.2% to 1.9% by varying the number of values per attribute on a million object database. Both algorithms were found to be largely insensitive to the varying number of values per attribute, BAA continuing to outperform Balke by orders of magnitude in terms of response times.

The results were similar in real-world datasets too, BAA continuously outperforming Balke by a factor of around 100. For the $ForestCover$ dataset, BAA response times were 12 seconds on the average, whereas Balke takes as much as 25 minutes to complete. On the much smaller $Census-Income$ dataset, BAA response time was seen to be around 2 seconds whereas Balke takes an average of 2.8 minutes.

Balke is different from BAA in that its memory requirement is unbounded (Ref. Section 3). Figure 6 plots the memory requirement in terms of the maximum number of candidates held in memory at any point in the execution of the algorithm. As expected, the memory requirement increases with the dataset size whereas the ratio of the memory requirement to the dataset size decreases with density, and stabilizes at around 23% of the dataset size. Thus, it seems reasonable to assert that the Balke can be employed only in such scenarios where the available memory is at least a quarter of the database size.

5.3.2 Analysis

The experiments in this section illustrate the effectiveness of the BAA algorithm. It does not require unbounded buffers (as Balke does) and is significantly better in terms of IO. BAA outperforms Balke in terms of response times on a wide range of c_r/c_s ratios and a wide range of data densities. Further, it is provably better than Balke in terms of IO costs (Ref. Section 3). This shows that BAA is the preferred algorithm for virtually all possible scenarios. Thus, we pick the BAA algorithm as the representative middleware algorithm to compare with other algorithms in subsequent sections.

5.3.3 IO Cost Lower Bounds

In this section, we analyze the IO costs of BAA as compared to the absolute lower bound of IO costs for middleware algorithms. Taking cue from the properties of middleware algorithms in Section 3.3, we develop an approach to compute the lower bound of IO costs for any middleware algorithm, and go on to show that BAA is empirically very close to the lower bound.

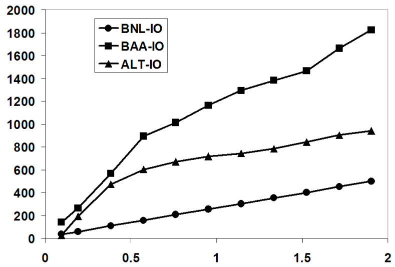

Figure 7: IO-Cost vs Density (Varying Data Size)

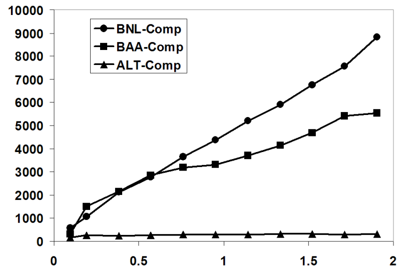

Figure 8: Comp Time (ms) vs Density (Varying Data Size)

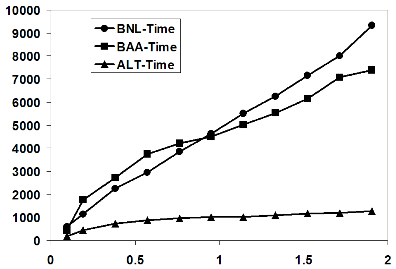

Figure 9: Response Time (ms) vs Density (Varying Data Size)

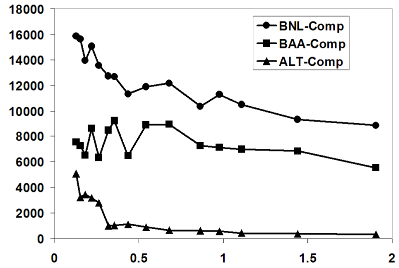

Figure 10: Comp Time (ms) vs Density (Varying #Values)

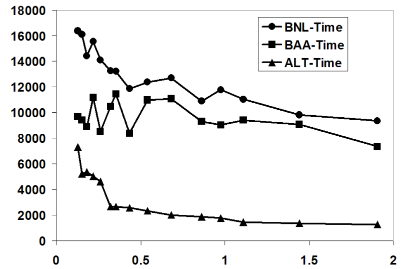

Figure 11: Response Time (ms) vs Density (Varying #Values)

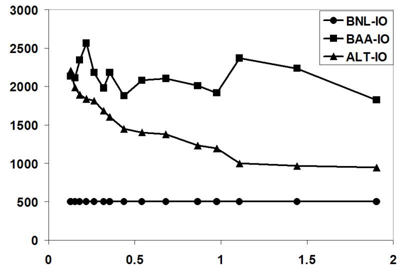

Figure 12: IO-Cost vs Density (Varying #Values)

The IO cost for a middleware algorithm on a given dataset and query is computed as the weighted sum of sequential and random item accesses. Any approach may be viewed as a sequence of accesses, each of which could be either sequential or random. However, Property 2 suggests that for any approach, there exists an approach which incurs the same or lesser cost, but does all its random accesses at the end. Property 3 further says that for any approach that decides to do only random accesses from thereon, the number of random accesses that need to be performed to claim correctness can be easily computed, and is independent of the order in which they are performed. The above properties apply to the IO optimal approach too. However, the IO optimal approach may choose to explore different lists to different depths (hence, not necessarily in round robin fashion) using sequential accesses, before it decides to do random accesses. Therefore, the following construction gives a lower bound on the IO cost of a middleware method for skyline: try all possible sequential scan depths in each of the input lists, and for each such valid combination (a combination is valid only if it contains an MSC), compute the cost of scanning until this depth plus the cost of the then absolutely necessary random accesses. The IO cost incurred by each combination can be computed by simulating such an approach. We explore the space of such combinations, simulating each combination, and identify the cost of the approach which corresponds to the IO optimal approach. Note that the outlined computation is not a real skyline algorithm in itself, but merely serves to determine the lower bounds of IO costs. However, as the space of all combinations is typically huge, we restrict the space further by exploring only those candidates which have each attribute as a multiple of 1000 (thus enabling us to complete such an analysis within a reasonable amount of time). To ensure a fair comparison, for this set of experiments, we restrict BAA to explore each list in chunks of 1000 items.

We compare the lower bound of IO costs against the IO cost incurred by the BAA algorithm on a uniform random dataset of 50000 items with 3 attributes, each having 35 values per attribute as well as a random subset of the *Census - Income* dataset with 50000 data items on 3 attributes (A subset was chosen because the lower bound computation is very compute intensive, making it infeasible to run on the whole dataset). The IO costs reported here are those averaged over 10 queries. We varied the c_r/c_s ratio from 1 to 10000. It may be noted that the access pattern of BAA changes with the ratio, since it tries to minimize the aggregate IO costs. Figure 13 shows that BAA closely follows the lower bound in terms of IO costs on the uniform random dataset. In the case of the real dataset, Figure 14 shows that BAA is around consistently 10% costiler in terms of IO costs than the lower bound when the ratio approaches commonly adopted values such as 10000. For both the datasets, Balke was found to be costlier by at least a couple of orders of magnitude (e.g., Figure 4) and hence, isnt shown in these graphs. This validates our claim in Section 3.3 and suggests that any better strategy would only give marginal improvement over BAA in terms of IO costs.

5.4 Performance of Skyline Algorithms

We analyze the performance of BAA and ALT algorithms against the BNL algorithm, the state-of-the-art algorithm that handles arbitrary similarity measures. The experiments are similar to those in Section 5.3.1, varying the density by separately varying the dataset size and the number of values per attribute. These experiments, being not on middleware, use a simulation of a disk-based implementation of BAA.

Figure 7 shows that the page IO costs of all algorithms increase with dataset size. The IO costs for the BNL and BAA algorithms are linear on the dataset size. It is inter-

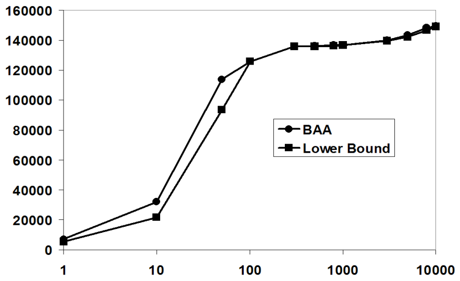

Figure 13: IO Cost vs Ratio (Synthetic Data)

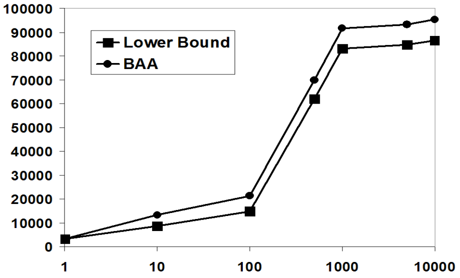

Figure 14: IO Cost vs Ratio (Census-Income Data)

esting to note that the rate of increase of IO costs of the ALTree algorithm decreases with density. The ALTree algorithm, as it employs a value based index, prunes out larger parts of the tree as the probability of finding objects with the closest value to the query on each attribute increases with the density of the dataset. It may be noted that the BNL, as it employs sequential scans of the database, outperforms the other algorithms in terms of IO costs. Although BNL and BAA algorithms spend more time in computation with increasing dataset size(Ref. Figure 8), BNL spends much more time than the BAA algorithm. Even though ALT and BAA have similar complexities, BAA has a higher candidate maintenance cost thus resulting in higher runtimes. Of particular significance is the behavior of the ALT algorithm whose computational costs remain unaffected by increasing dataset size. This is because the increased pruning in higher densities significantly helps in reducing computational costs, as lesser number of objects need to be checked for inclusion in the skyline. Figure 9 plots the overall response times for the various algorithms. The ALT algorithm overwhelmingly outperforms the BAA and BNL algorithms (which closely follow each other) as it leverages the increased density to exclude database objects without even seeing them once.

Further, we analyzed the behavior of various algorithms when the change in data density is caused by changing number of values per attribute on a million object dataset. Figure 12 shows that the IO cost of the ALT algorithm decreases with increasing densities, as it is able to leverage the higher densities to affect better tree pruning. The BNL algorithm has a constant IO cost, as it performs sequential scans of the database, whose size remains unaffected with change in the number of values per attribute. BNL performs better than ALT in terms of IO costs in lower densities; but, the gap narrows with increasing density. This shows that BNL would outperform our algorithms on extremely sparse data. Figure 10 shows that BNL has a high computational cost, whereas the ALT spends much lesser time in computations, especially, in higher densities. Much on expected lines, Figure 11 shows that the ALT outperforms both of the other algorithms by a large margin in terms of response times. The BAA algorithm is fairly passive to varying number of values per attribute (similar to Section 5.3.1).

The better performance of the BAA and ALT algorithms relies on their ability to exclude objects from the skyline without even seeing them once. This is in sharp contrast to BNL, which needs to see the entire dataset at least once. To analyze the effectiveness of such pruning strategies, we analyze BAA and ALT based on the fractions of sorted lists visited and the fraction of nodes visited respectively. ALT displays a sharp drop in the fraction of the tree visited with increasing density (Figure 15) and is able to manage by visiting as less as 9% of the tree at high densities. BAA consistently visits a much lesser fraction ($4-5\%$) of the sorted lists. However, as the objects seen in the various lists may be different, the number of objects visited may be as high as BAA-Ratio*m (i.e., $20-25\%$ in this case).

5.4.1 Performance Analysis on Real Datasets

Real world datasets are usually very skewed (non-random) and thus may be significantly different from the synthetic random datsets, on which we reported results in the previous sections. As mentioned in Section 5.2, we use two datasets with widely varying densities. It may be noted that the skyline size increases with the sparsity of the dataset, leading to reduced possibilities of pruning; thus, sparse datasets are adverse scenarios for the BAA and ALT algorithms. Figure 17 plots the response times for disk based implementations of the algorithms against varying cache sizes on the $ForestCover$ dataset, which is very sparse (density of 0.04%). ALT outperforms both BNL and BAA by close to a factor of 4 consistently, which is tremendous, given that sparse data and low cache sizes present an unfriendly scenario to ALT. This shows that the ALT is able to exploit the skew in the data to its advantage. The BAA algorithm improves its response times with increasing cache size at low densities, primarily due to being able to preserve the objects in the same pages as those objects on which random accesses were performed. As expected, that effect is not so pronounced at higher densities. Thus, at reasonable cache sizes, both the BAA and ALT algorithms outperform BNL with ALT having remarkable response times close to 200 milliseconds. The high density of the $Census-Income$ dataset is favorable to BAA and ALT algorithms. Similar to the observations for ForestCover, the ALT is able to use the skew in the dataset to deliver very good response times (Ref. Figure 16), consistently returning the skyline within 14 milliseconds. The BAA algorithm is able to utilize the high density to stop earlier and has a response time of around 200 milliseconds. The BNL algorithm is outperformed by orders of magnitude by both the BAA and ALT algorithms.

6. CONCLUSIONS AND FUTURE WORK

We have presented two algorithms, namely the BAA algorithm and the online AL-Tree based algorithm for skyline retrieval with arbitrary similarity measures. We present a general framework for middleware skyline algorithms and analyze them formally. Through a detailed set of experiments on various types of synthetically generated data and

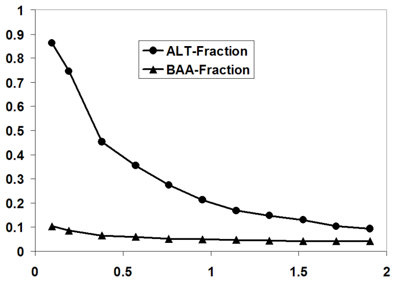

Figure 15: Fractions of Datasets visited (Varying Data Size)

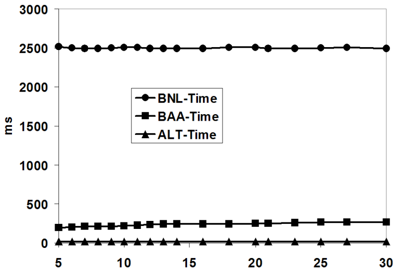

Figure 16: Response Time vs Cache Size (Census Data)

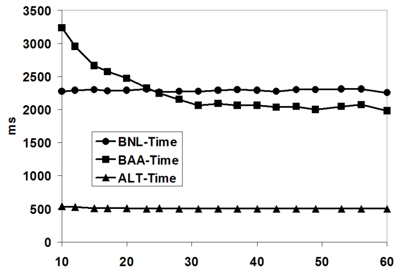

Figure 17: Response Time vs Cache Size ForestCover dataset

two real datasets, we draw the following conclusions.

- BAA is the preferred middleware algorithm (over Balke) for virtually all possible scenarios. Its IO costs are empirically found to be very close to the absolute lower bound of IO costs for any middleware algorithm, and when nothing is known about the attributes and their dependencies, it seems unlikely that BAA can be provably improved upon, in terms of IO costs.

- Our BAA and AL-Tree based skyline algorithms outperform the existing BNL approach significantly in computation cost and overall cost. In presence of a very fast CPU with very high random access costs, BNL may be of use.

- The AL-Tree based skyline algorithm outperforms both BNL and the BAA middleware algorithm by orders of magnitude in most usual settings.

BAA has a high candidate maintenance overhead. An interesting future work will be to devise more efficient candidate maintaining strategies for the skyline algorithms for middleware. It will be interesting to analyze if sequential access scheduling approaches similar to those used for top-k algorithms [4] can be adapted to improve skyline algorithms. For the AL-Tree based approach, investigating whether any specialized ordering of the attributes enables faster processing, is also a potential future work. Since the AL-Tree based approach is able to prune out large sections of the tree, better packing approaches could help in optimizing the IO costs. $\beta(X)$ could be redesigned to return multiple (bounded by a constant) objects for each node X, leading to better pruning.

7. REFERENCES

[1] How fast is your disk? *http://www.linuxinsight.com/how_fast_is_your_disk.html*, January 2007.

[2] A. Asuncion and D. Newman. UCI machine learning repository, 2007.

[3] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, pages 256–273, 2004.

[4] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*. ACM, 2006.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[6] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, 2003.

[8] W. Chung, Gray and Horst. Windows 2000 disk io performance. *Microsoft Research TR*, June 2000.

[9] K. Deng, X. Zhou, and H. T. Shen. Multi-source skyline query processing in road networks. In *ICDE*, 2007.

[10] P. Deshpande, Deepak, and K. Kummamuru. Efficient online top-k retrieval with arbitrary similarity measures. In *EDBT*, 2008.

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

[12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[13] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, 2005.

[14] K. Goh, B. Li, and E. Chang. Dyndex: A dynamic and nonmetric space indexer. In *ACM Intl. Conference on Multimedia*, 2002.

[15] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286. Morgan Kaufmann, 2002.

[16] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD Conference*, 2003.

[17] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[18] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310. Morgan Kaufmann, 2001.

[19] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *ICDE*, page 65. IEEE Computer Society, 2006.

[20] S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu. Efficient skyline query processing on peer-to-peer networks. In *ICDE*, pages 1126–1135, 2007.

[21] P. Zesula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach.* Springer, 2005.