Efficient Reverse Skyline Retrieval with Arbitrary Non-Metric Similarity Measures

Prasad M. Deshpande Deepak P
IBM Research - India, Bangalore
India Research Lab, Bangalore
{prasdesh,deepak.s.p}@in.ibm.com

ABSTRACT

A Reverse Skyline query returns all objects whose skyline contains the query object. In this paper, we consider Reverse Skyline query processing where the distance between attribute values are not necessarily metric. We outline real world cases that motivate Reverse Skyline processing in such scenarios. We consider various optimizations to develop efficient algorithms for Reverse Skyline processing. Firstly, we consider block-based processing of objects to optimize on IO costs. We then explore pre-processing to re-arrange objects on disk to speed-up computational and IO costs. We then present our main contribution, which is a method of using group-level reasoning and early pruning to micro-optimize processing by reducing attribute level comparisons. An extensive empirical evaluation with real-world datasets and synthetic data of varying characteristics shows that our optimization techniques are indeed very effective in dramatically speeding Reverse Skyline processing, both in terms of computational costs and IO costs.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search Process*

1. INTRODUCTION

The skyline operator is a useful tool in multi-criteria similarity search [4]. It is used to assess whether an object *dominates* another with respect to a query object. An object A is said to dominate another object B with respect to a query object if A is at least as similar as B to the query object on all dimensions under consideration and strictly more similar in at least one dimension. Consider the problem of searching for cars with preference for cheaper rates and high mileage (i.e., a hypothetical query object, a car that provides infinite mileage and can be bought for zero units of currency); a car that is cheaper than another and also provides higher mileage would dominate the other. The skyline result set for a query is computed as the set of objects that are not dominated by any other. In our example, we would choose not to suggest the latter car since it is not better than the former on any attribute considered. This is notably different from the top-k set, where the k objects that are closest to the query are retrieved. Assessing the top-k similar objects requires, for every object, a single score that is directly related to its similarity to the query; this score is often computed as a monotonic aggregation function (e.g., a weighted sum) of the similarity to the query based on the various dimensions considered. The skyline set, on the other hand, does not depend on any aggregation function (such a function would be quite hard to specify for many scenarios). Interestingly, for every point in the skyline, there exists a monotone aggregation function that is maximized at that point. Thus, the skyline set does not contain any object that is not the best according to some possible monotonic aggregation function.

The Reverse Skyline [9] (RS) set for a particular query, is defined as the set of objects that have the query in *their* skyline set. Informally, an object in the database is part of the RS for a query, if the query is in the Skyline set for that object. This is related to the Reverse Nearest Neighbor (RNN) query [15, 25, 27] that retrieves all objects that have the query object as it's nearest neighbor. However, the RNN, similar to the top-k query, is assessed with respect to a pre-specified aggregation function (most commonly, a monotonic aggregate). *The Reverse Skyline set is the union of the RNN set across all possible specifications of monotonic aggregation functions.* RS is a convenient replacement for RNN in cases where one specific monotonic aggregate is hard to specify. In the hotel example, even if the user is sure about his preference to the proximity criterion than the tariff, the correct weighting to use may not be obvious.

The car search example (adapted from [9]) is evidently simplistic since factors that influence the choice of a particular car are not just price and mileage. Other factors include *Manufacturer*, *type of fuel used* (e.g., choices may vary from diesel and petrol to electric and LPG), *available colors*, set of safety features and entertainment equipments. The diversity of attributes makes it difficult to specify a weighting scheme (and thus an aggregation function) across them to quantify (by means of a single score) how well a user is likely to like a specific car. When the set of preferences of a user is specified in the same manner as a car is specified, a user would intuitively prefer a car over another if the latter is only at most as similar (as the former) on *all* preferences under consideration. This being precisely the notion of *domination* in skyline, the choice of cars to suggest to the user, in a recommender system, would be from the skyline set with respect

EDBT 2011, March 22–24, 2011, Uppsala, Sweden.

to the user preferences. The *influence* of a car can then be assessed based on their Reverse Skyline (RS) set; the RS set provides the set of users for which the car is likely to be a good choice. Such influence assessments are critical to a dealer of pre-owned cars; he/she may want to source more of the *influential* cars since they are more likely to be bought by customers.

We encountered the problem of having to identify the RS set, in the context of business continuity planning for a service delivery organization that employs thousands of system administrators to manage hundreds of thousands of servers. *System Administrators* are specialists in managing servers and troubleshooting problems in them. Over time, they gain expertise in solving problems specific to certain software, operating systems and specific type of hardware. The expertise of a system administrator would then be represented as a vector of such acquired expertise grouped into categories (e.g., operating system, network type etc.). Servers could also be mapped to such a space with appropriate values for the various categories. Now, the choice of admins for a particular server would be from the skyline set for the server. Highly influential admins (those who are suitable for many servers, due to having a larger RS set) are then critical to the business; heavily skewed influence distribution among admins and attrition of highly influential admins are all causes of concern due to obvious reasons. We use the server scenario as the running example in the rest of the paper.

A similar scenario arises in choosing retail customers to send promotional mailing to, for a new offer on a particular product. Since the retail company would want to choose customers who are most likely to respond positively, the RS set is a good choice being the set of users whose preference to the product is not dominated by other products.

1.1 Non-metric Spaces

Points in a metric space need to satisfy the triangle inequality, i.e., $d(x, y) + d(y, z) \geq d(x, z)$. However, many real life scenarios involve categorical and set-valued attributes, where the similarity measures may be non-metric. For example, attributes of a server could include the DB installed, operating system and network card details (categorical) and the set of software installed on a server (set-valued). Similarities between operating systems come from domain knowledge with the domain expert defining the similarities for each pair of operating systems. While filling in a matrix of pairwise operating system similarities as assessed using experience and domain knowledge, the domain expert cannot be expected to comply with the metric requirement of triangle inequality. Such similarities, hence, may not even be *arbitrary metric* [1]. We will see in Section 2 that such non-metric similarity measures are often necessary to model the conceptual notion of similarity. Attributes where the similarity between values is non-metric do not have a total ordering among their values. For example, there is no global ordering of the operating systems available. An ordering of values for each attribute can be arrived at only when values are considered with respect to a chosen value for the attribute (typically, the value of the query object).

Multi-dimensional indexes such as R-tree are applicable only when there is an ordering of values for each attribute (Euclidean space). However, in the case of non-metric similarities, different query objects may take different values for an attribute leading to different orderings. It is not possible to create a single index that can be used for different objects under consideration; this renders such an approach impractical. M-tree [8] avoids the need for a Euclidean distance, but requires the triangle inequality property. In the absence of triangle inequality property, the similarity between a pair of objects cannot be reasoned about by knowledge about their separate similarities to a third object. *This makes metric space indexes (e.g., R-Tree [14], kd-tree [3], M-Tree [8] etc.) for similarity search inapplicable.*

1.2 Our Contributions

In this paper, we address the problem of Reverse Skyline retrieval on databases of objects composed of multiple attributes. We assume a setting where the dissimilarity between pairs of values of the same attribute may be non-metric i.e., our techniques do not require that the dissimilarities confirm to a criterion such as triangle inequality.

Without the metric assumption, the naive method for Reverse Skyline retrieval requires scanning the database, once for each object O, to confirm the absence or presence of another object O' that could dominate the query object Q with respect to O. Since the presence of one such O' would exclude (i.e., *prune*) O from the Reverse Skyline set, we could skip scanning the rest of the database as soon as one such object is found. However, since objects in the Reverse Skyline set are characterized by the absence of any such O', a full database scan is necessitated for such objects. This requires n partial/full scans and has a worst case computational complexity of $\mathcal{O}(n^2)$ where n is the number of objects; such an approach evidently incurs very high IO costs too.

In this paper, we devise approaches that employ the following optimizations, to develop more efficient algorithms for Reverse Skyline retrieval:

1. Considering objects in contiguous batches (according to their placement in disk), to optimize on IO costs.

2. Pre-sorting of the database to ensure that objects and those objects that they could prune be kept close to each other on disk; this increases the likelihood of objects and pruners being in the same disk page, thus reducing IO and computational costs.

3. Group-level reasoning and early pruning using an in-memory AL-Tree [10] to enable faster processing.

Our main contribution is the technique to use group-level reasoning and early pruning to speed-up the RS search; we also present how techniques that use a subset of the above optimizations fare, thus exposing the utility of each. We present an extensive empirical evaluation on real and synthetic datasets illustrating the efficiency of the tree based approach, and confirm it as the algorithm of choice for virtually all possible scenarios including those where the user is given an option to choose a subset of attributes to perform the Reverse Skyline search on.

2. RELATED WORK

The need for *non-metric similarity functions* has been argued in [13], that says that the triangle inequality property is too restrictive to model the (dis)similarities as perceived by humans. [19] opines that the conceptual notion of similarity is centered around various aspects, and that different

aspects may be selected for comparing different pairs of objects. Such choices of aspects to compare pairs cannot be pre-determined; this makes metric assumptions inapplicable. Further, non-metric similarity measures have been found to be useful in similarity search in various kinds of data ranging from images to object trajectories [26, 5]. [24] points out specific cases in which each of the metric properties (viz., reflexivity, symmetry and triangle inequality) may not be intuitively satisfied. Similarity search on arbitrary non-metric similarity measures has attracted recent attention [10, 21, 20].

The *Skyline* operator, upon which the concept of Reverse Skyline is built, was analyzed in detail for the first time in [4]. Two flavors of the skyline problem have been studied in literature: (1) Retrieving the skyline for the database [22], (2) Retrieving the skyline from the database for a given query object (often called *dynamic skyline*). The former problem is applicable only in a database where *all* attributes are from ordered domains and there is an obvious intuitive query object. In the example of choosing a hotel, such a query object would be a hotel situated right at the conference center and has a hypothetical tariff of 0 units of currency. Among the algorithms that address the more general problem of query based skyline retrieval (*dynamic skyline*) and can handle arbitrary non-metric similarity measures are Block Nested Loops (BNL), Divide & Conquer [4] and SkylineDFS [21]. The Skyline retrieval problem has also been addressed in the popular middleware setting [12], where the recent BAA algorithm [21] has been found to be effective over others [2].

The *Reverse Skyline* query was introduced in [9] wherein the BBRS and RSSA algorithms for metric-space data were proposed. Techniques for metric-space Reverse Skyline retrieval over uncertain data [17, 18] and streaming data [29] have also been proposed. *To the best of our knowledge, this is the first work addressing the problem of Reverse Skyline retrieval under arbitrary non-metric similarity measures.*

Since the *skyline* comprises of all those objects that are the nearest neighbor according to some monotonic aggregation function, the Reverse Skyline query is a generalization of the Reverse Nearest Neighbor (RNN) query where all possible monotonic aggregation functions are considered. Although RNN has been well-studied in metric spaces [15, 23], approaches that work on arbitrary metric [1] and non-metric spaces [6, 28] have been scant. The only approaches [6, 28] that consider RNN processing with arbitrary similarity measures require specifying the aggregation function at index creation time (as against query time), making it impractical to generalize them for Reverse Skyline processing.

3. PROBLEM DEFINITION

We will now define the problem formally. Let $\mathcal{D}$ be the set of objects in the database, each of them having m attributes. The dissimilarity function d_i, for the i^{th} attribute, is a function $d_i : \mathcal{A}_i \times \mathcal{A}_i \rightarrow \mathcal{R}$ where $\mathcal{A}_i$ is the domain of the i^{th} attribute. 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 value of the i^{th} attribute of object O. For notational convenience, we will use $d_i(X, Q)$ instead of $d_i(v_i(X), v_i(Q))$ whenever the context is clear. The second condition above ensures that duplicates (i.e., objects that have the same value for all attributes) do not dominate one another. The skyline from a database $\mathcal{D}$ for an object X, denoted as $S_{\mathcal{D}}(X)$ is the set of all objects that have not been dominated by another object with respect to X

$$S_{\mathcal{D}}(X) = \{Y | Y \in \mathcal{D} \wedge (\nexists Z \in \mathcal{D}, Z \succ_X Y)\}$$

Definition 1. **Reverse Skyline Problem:** Given a query Q, an object X from $\mathcal{D}$ is in the Reverse Skyline of Q (denoted as $RS_{\mathcal{D}}(Q)$) *iff* Q is in the skyline of X.

$$RS_{\mathcal{D}}(Q) = \{X | X \in \mathcal{D} \wedge Q \in S_{\mathcal{D} \cup \{Q\}}(X)\}$$

Note that Q need not belong to the database $\mathcal{D}$. We can alternatively denote the same set as:

$$RS_{\mathcal{D}}(Q) = \{X | X \in \mathcal{D} \wedge (\nexists Y \in \mathcal{D}, Y \succ_X Q)\}$$

Thus, to confirm the membership of an object X in $RS_{\mathcal{D}}(Q)$, one need not explicitly compute the set $S_{\mathcal{D}}(X)$ and check for membership of Q in it; it is only necessary to verify whether there exists an object Y that dominates Q with respect to X. In particular, one can discard X from $RS_{\mathcal{D}}(Q)$ on finding one such Y since that confirms the absence of Q in $S_{\mathcal{D}}(X)$. We refer to such a Y that dominates Q with respect to X as a *pruner* of X, since its presence excludes X from $RS_{\mathcal{D}}(Q)$.

4. REVERSE SKYLINE ALGORITHMS

The simplest approach for *Reverse Skyline* retrieval would involve checking, for every object X, whether there exists a Y that dominates the query with respect to X. Such an algorithm is outlined in Algorithm 1. Assuming that data objects are arranged contiguously on disk, this approach incurs many scans of the database (upto $|\mathcal{D}|$ full scans) although some scans can end much earlier upon discovery of a Y.

Alg. 1 ***Naive***

```
1. Result = φ
2. ∀X ∈ D
3. pruned = false
4.       ∀Y ∈ D, Y ≠ X
5.             if (Y ≻_X Q)
6.                   pruned = true; break;
7.       if (!pruned) Result = Result ∪ {X}
```

In this paper, we propose the following optimizations to Reverse Skyline retrieval and propose algorithms to use such optimizations in an effective manner to improve performance.

1. **Block-based Accesses:** Data on disk is accessed in term of pages. Even if only one specific object needs to be retrieved, typical systems retrieve the entire page that the object resides in, from disk. Thus, making full use of objects in a page, when accessing it, would help reduce IO costs. Accessing disk pages sequentially would help further reduce IO costs since sequential IOs are cheaper than random IOs.

2. **Pre-Sorting:** An arrangement of objects in disk such that objects that could prune them from the $RS_{\mathcal{D}}(Q)$ set be found closer on disk would reduce the IO costs (objects may be pruned upon doing fewer page IOs) as well as computational costs (since objects get pruned earlier while doing sequential accesses).

3. **Group-level Reasoning and Early Pruning:** Consider an object X, having a value a for the first attribute A_1. No object that takes value b for A_1 can dominate Q with respect to X if b is farther away from a than is $v_1(Q)$, according to $d_1(.,.)$ (since domination requires it to be at least as close to X on *each* attribute, as the query Q). Such group-level reasoning on objects that share the same value for a specific attribute, eliminates the need to check whether each of them can dominate Q separately; this could lead to significant speed-ups.

   The search for a pruner for any object X can stop as soon as the first pruner is found (see Line 6 in Algorithm 1). Thus finding a pruner early can significantly save on computational costs. Group level reasoning enables removing groups of useless candidates. We can further speed up processing by ordering the checks to guide the search along more promising paths.

In the rest of this section, we develop algorithms to use such optimizations to speed up Reverse Skyline processing. While our main contribution is the technique to do group-level reasoning and early pruning (leading to many factors of speed-up as we will see in Section 5), we analyze how each of the above optimizations contribute to the speed-up.

We will use the dataset in Table 1 and the distance functions in Figure 1 as a running example for the rest of this paper. This hypothetical dataset of servers has three attributes, the *Operating System*, *Processor* and the *Database*, taking three, two and three values respectively. It can be seen that $d_1(.,.)$ is non-metric; $d_1(MSW, SL)$=1.0 is greater than the sum of the distances $d_1(MSW, RHL)$=0.8 and $d_1(RHL, SL)$=0.1. The dissimilarities between *Processors*, d_2 is specified as $d_2(AMD, AMD) = 0$, $d_2(Intel, Intel) = 0$ and $d_2(AMD, Intel) = 0.5$. The $d_3(.,.)$ function lists dissimilarities between the various DBs. Now, consider a query object $Q = [MSW, Intel, DB2]$; the fifth column in Table 1 lists the membership of each object in $RS_{\mathcal{D}}(Q)$ for this query. For each object marked as not in the result, the pruners are also listed. For example, it is possible to prune $O_2 = [RHL, AMD, Informix]$ by $O_1 = [MSW, AMD, DB2]$, since O_2 is closer than the query to O_1 on the second attribute (i.e., $d_2(AMD, AMD) < d_2(AMD, Intel)$ from Figure 1) and at the same distance as the query to O_1 on the remaining. Thus, O_1 is listed a pruner for O_2. The result set for this query is $\{O_3, O_6\}$, since they do not have any pruners.

d_1	MSW	RHL	SL
MSW	0.0	0.8	1.0
RHL	0.8	0.0	0.1
SL	1.0	0.1	0.0

d_3	Informix	DB2	Oracle
Informix	0.0	0.5	0.9
DB2	0.5	0.0	0.4
Oracle	0.9	0.4	0.0

Figure 1: Distance Functions.

Id	OS Name	Processor	DB Name	in $RS_{\mathcal{D}}(Q)$?
O_1	MS Windows (MSW)	AMD	DB2	$\times\{4\}$
O_2	RedHat Linux (RHL)	AMD	Informix	$\times\{1, 4, 5\}$
O_3	SuSE Linux (SL)	Intel	Oracle	✓
O_4	MS Windows (MSW)	AMD	DB2	$\times\{1\}$
O_5	RedHat Linux (RHL)	AMD	Informix	$\times\{1, 2, 4\}$
O_6	MS Windows (MSW)	Intel	DB2	✓

Table 1: Sample dataset and RS for $Q = [MSW, Intel, DB2]$

Alg. 2 ***BRS***

```
1. /* first phase */
2. while(D has not been fully processed)
3.     load next batch of objects into memory
4.     ∀X loaded into memory
5.         ∀Y loaded into memory, Y ≠ X
6.             if(Y ≻_X Q)
7.                 mark X as pruned; break;
8.     write unpruned objects into disk
9. /* second phase */
10. let R = objects written to disk in first phase
11. while(R has not been fully processed)
12.     load next batch of objects(R') into memory
13.     for each disk page of D
14.         load the objects in that page D', into memory
15.         ∀X ∈ R'
16.             ∀Y ∈ D', Y ≠ X
17.                 if(Y ≻_X Q)
18.                     remove X from R'; break;
19.     output R'
```

4.1 Block Reverse Skyline (BRS)

When the available memory is sufficient to hold the entire database, Algorithm 1 may be employed in memory after a single scan to load the entire database into memory. However, typical databases are too large to fit in memory. Now, we describe an approach that performs block-wise accesses to reduce IO costs. Our block based approach, BRS is illustrated in Algorithm 2 and operates in two phases.

First Phase: BRS, works in batches and loads as much of the database as possible, into memory, during each such batch (line 3). Once such a batch is loaded, objects within it, that have pruners within the batch itself (verified by doing intra-batch pairwise comparisons) are marked (lines 4-7). The remaining objects are then written out into a separate area on disk (line 8). When the entire database has been processed in such batches, the writing area would have a superset of $RS_{\mathcal{D}}(Q)$ (denoted as $\mathcal{R}$ in line 10). This is because only intra-batch pruning has been performed; objects for whom all pruners were outside its own batch would still find themselves in the writing area. The *false positives* among them are then filtered in the second phase.

Second Phase: This phase also works in batches, by loading as many of objects from the first phase results as possible, into memory (line 12). For each such batch of first phase results, objects from the original database are read page by page (lines 13-14) and any objects that they can prune from among the first phase results are excluded (lines 17-18). At the end of the complete sequential scan of the original database to affect such pruning, only true positives remain. These form a subset of $RS_{\mathcal{D}}(Q)$, and are output

(line 19). Upon processing all of the first phase results in such batches, we would have output the entire result set. One page memory is used to scan the original database and the rest of the memory is used to load the first phase results.

Consider a hypothetical page size that can hold only one object, and a memory size of 3 pages, on our running example. Thus, the first batch in the first phase would consider the first three objects. O_2 is pruned by O_1 by intra-batch pruning, whereas O_1 and O_3 are carried forward to the next phase. Similarly, the intra-batch pruning in the second batch can only prune O_5. Thus, the set of first phase results, $\mathcal{R}$ would be $\{O_1, O_3, O_4, O_6\}$. In the second phase, since we want to leave one page for scanning the database, we would process the partial result in batches of at most 2 pages (and hence, 2 objects), $\{O_1, O_3\}$ and $\{O_4, O_6\}$. Among them, O_1 would get pruned when O_4 is encountered in the scan of the database, whereas O_3 would be output. Similarly, O_5 is the output in the second batch, thus completing the full result set (i.e., $\{O_3, O_6\}$).

IO Costs: The first phase is a simple sequential scan of the database, in addition to random accesses to go and write out the results at the end of processing each batch and to return to resume the scan of the next batch. The second phase incurs one sequential scan of the database for each batch of first phase results considered, and a random access to return to resume scanning the next batch.

4.2 Sort Reverse Skyline (SRS)

In the BRS approach, an object not in the result could be either pruned in the first phase by objects in the same batch or in the second phase where we scan for pruners in the entire database. Pruning more number of objects in the first phase itself would reduce the computational cost (since we only compare them to same batch objects in the first phase) and the IO cost (since fewer objects have to be processed in the second phase, which could save on database scans).

It can be observed that the chance of an object pruning another increases if they share the same value for some of their attributes. This is due to the fact that the dissimilarity of a value and itself is intuitively zero (i.e., $d_i(x, x)$=0) for most dissimilarity functions. Thus, if two objects take the same value for an attribute, they have the smallest possible dissimilarity (i.e., 0) on that attribute. Whether one can prune another *with respect to any query* is decided based *only* on the other attributes. Since the first and sixth objects in our running example take the same value for the OS and DB attributes, the following is the condition upon which O_6 can be pruned by O_1, for any query Q:

$$d_3(Intel, AMD) < d_3(Intel, v_3(Q))$$

Reducing the number of attributes upon which pruning depends, increases the chances of pruning since only fewer conditions need be satisfied. Thus, *it is useful to keep objects that share the same value for many attributes together on disk as they could be part of the same batch and get pruned in the first phase.*

In order to achieve this, we employ a simple mechanism, a *simple multi-attribute sort* of the database.The database is ordered according to the first attribute values , and the objects that take the same value for the first attribute are ordered according to the second attribute values and so on. The actual ordering among different values of an attribute is immaterial while sorting. This sorting is only to ensure that objects that take the same values for the attributes are clustered together. After such a sort, the order of object ids in our running example would be $\{O_1, O_4, O_6, O_2, O_5, O_3\}$. It may be noted that this serves a different purpose than the sort in *Sort-First-Skyline* [7], where the sort is done on numerical attributes to ensure that an object cannot be pruned by an object occurring after it. In our case, objects beyond one can potentially prune it and vice versa, since there is no inherent order among the values due to the non-metric nature. This sort is a one-time effort, done as a pre-processing step, using any of the external sorting algorithms[1].

Approach	1^{st} Phase Pruning	$\mathcal{R}$	2^{nd} Phase Pruning	$\#2^{nd}$ Phase Batches
BRS	$\{O_2\}, \{O_5\}$	$\{O_1, O_3, O_4, O_6\}$	$\{O_1\}, \{O_4\}$	2
SRS	$\{O_1, O_4\}, \{O_2, O_5\}$	$\{O_3, O_6\}$	$\{\}$	1

Table 2: Performance on Running Example

The query time processing for SRS is very similar to the BRS approach, so we haven't listed it explicitly. The only difference is that while searching for pruners of X (lines 5-7), we consider Y in the order of their similarities to X, to enable finding the pruner early. Thus, for each X we first consider the objects immediately next to it in either direction of the sorted order, followed by objects at separation distance of 2 in the sorted order and so on.

To facilitate understanding of the performance of SRS on our running example, we reproduce the pruning relationships ($a \rightarrow B$ suggesting that a can prune each object in B): $\{O_1 \rightarrow \{O_2, O_4, O_5\}$, $O_2 \rightarrow \{O_5\}$, $O_4 \rightarrow \{O_1, O_2, O_5\}$, $O_5 \rightarrow \{O_2\}\}$. With a memory size of 3 pages, one per object, and the sorted ordering $\{O_1$,O_4,O_6,O_2,O_5,$O_3\}$, $\{O_1$,O_4,O_2,$O_5\}$ are pruned by intra-batch prunings. Thus, we are left with $\mathcal{R}$=$\{O_3, O_6\}$. These are considered in one batch in the second phase; however, since they do not have pruners in $\mathcal{D}$, they get output. Table 2 contrasts the approaches on the running example. Pre-sorting improves 1^{st} phase pruning and SRS is able to execute the 2^{nd} phase in one batch, incurring one less database scan as compared to BRS.

4.3 Tree Reverse Skyline (TRS)

Consider O_1 ($[MSW, AMD, DB2]$) and O_6 ($[MSW$, $Intel$, $DB2]$) in our running example and the process of checking whether O_1 would prune O_6 for our query, [MSW, $Intel$, $DB2$]. Our objective is to check whether the following pruning condition holds:

$$\forall_i, d_i(O_6, O_1) \leq d_i(O_6, Q) \wedge \exists_i, d_i(O_6, O_1) < d_i(O_6, Q)$$

While checking whether this condition holds across attributes, we could abort as soon as we find an attribute j where $d_j(O_6, O_1) > d_j(O_6, Q)$ since that entails that O_1 would not prune O_6 (the above condition can no longer be satisfied). If we start examining attributes from the first, such a condition arises at the 2^{nd} attribute since $d_2(Intel, AMD) > d_2(Intel, Intel)$. Thus, we process this check in 2 comparisons, instead of 3 (the number of attributes). It can be observed that any other object that shares the same first two attribute values as O_1, would also not satisfy the pruning condition. In the example, O_4 shares the same prefix of length 3 as O_1. Upon performing two comparisons, and failing at the second, we would conclude that no object that shares the same length 2 prefix (in our case, neither O_1 or

[1] http://en.wikipedia.org/wiki/External_sorting

Alg. 3 ***TRS***

```
1. /* first phase */
2. while(D has not been fully processed)
3.       load next batch of objects from D into tree M
4.       ∀c loaded into memory
5.            if(IsPrunable(c, M \ c))
6.                 mark c as pruned
7.       write unpruned objects into disk
8. /* second phase */
9. let R = objects written to disk in first phase
10. while(R has not been fully processed)
11.      load next batch of objects from R into tree M
12.      for each disk page of D
13.           load the objects in that page D', into memory
14.           ∀e ∈ D'
15.                M = prune(e, M)
16.      output objects remaining in M
```

O_4) would prune O_6. Thus, our check for whether either of O_1 or O_4 would prune O_6 is complete in 2 checks, as against 4 in the object-by-object approach. To achieve this, we need to organize the objects according to their prefixes, so that we can use common checks to reason about groups of objects. By using a structure where objects share paths from the root, as long as they share attribute values, we optimize on the number of comparisons. We expand this illustrative case to a full-fledged approach herein.

The overall framework of the approach remains similar to *Block-RS* and is presented in Algorithm 3; similar to *SRS*, *TRS* works on a sorted dataset. The crux of the optimization lies in the pruning, different kinds of which occur in the first and second phases (line 5 and 15); we describe these in detail.

We facilitate group level reasoning based on shared prefixes to aid pruning using an in-memory variant of the AL-Tree [10]. Consider the database $\mathcal{D}$ and the chosen ordering of attributes $OS, Processor\ and\ DB$. The AL-Tree for $\mathcal{D}$ and the chosen ordering is then precisely the prefix tree[2] for the ordered database. For usage in Reverse Skyline processing, we are not concerned with sibling ordering and disk-packing and will use it as an in-memory data structure. To allow for duplicates, we will store a count in each leaf node denoting the number of objects that take the specific choice of values for various attributes as is denoted by the leaf. In our running example where we consider objects in batches of 3 each, the prefix trees generated are illustrated in Figure 2.

In each of the phases when we load objects, we incrementally build up a prefix tree of loaded objects. It may be noted that we are still dealing in batches of objects; a tree is built per batch, and at no point are we holding the tree corresponding to the entire database in memory. Adding an object to a prefix tree is simple; we follow the path in the tree (starting from the root) corresponding to the sequence of values taken by the object, creating a new child when a child is not already available for a particular attribute value. Upon reaching an existent leaf (in which case an exact duplicate of that object is already present in the tree), the counter at the leaf is incremented.

Pruning operations happen in both the phases:

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

Alg. 4 ***IsPrunable***$(c = [c_1, \ldots, c_m]$***,tree*** $M)$

```
1. Stack = φ
2. Root.FoundCloser = false
3. Stack.push(Root)
4. while(Stack ≠ φ)
5.       s = Stack.pop()
6.       if(s.isLeaf() ∧ s.FoundCloser)
7.            return true
8.       ∀p ∈ s.children,
         (in increasing order of number of descendants)
9.            if(d_{p.level}(c, p) ≤ d_{p.level}(c, q))
10.                p.FoundCloser = s.FoundCloser
                              ∨(d_{p.level}(c, p) < d_{p.level}(c, q))
11.           Stack.push(p)
12. return false
```

- **First Phase:** After a batch of objects from $\mathcal{D}$ has been loaded into a tree, we assess, for each object, whether it could be pruned by other objects in the tree. The $IsPrunable(c, M)$ function determines whether c can be pruned by objects in M. During this operation, we avoid the obvious special case of the occurrence of c in M from pruning itself by removing c from M before calling $IsPrunable$ (line 5).
- **Second Phase:** When a batch of first phase results have been loaded into a tree, we identify, for each object in $\mathcal{D}$, the set of objects in the tree that it could prune. The $Prune(e, M)$ function modifies M to exclude all objects that can be pruned by e (other than e itself, if present in M).

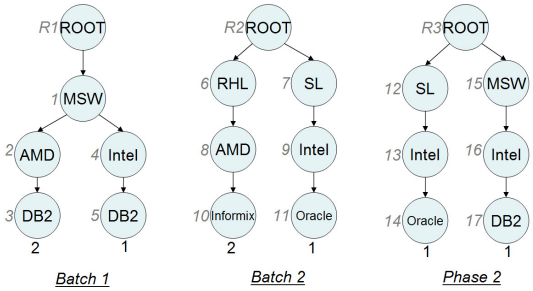

Figure 2: Trees for 1^{st} phase batches and 2^{nd} phase.

IsPrunable(c,M): We now describe our approach to check whether an object $c = [c_1, c_2, \ldots, c_m]$ can be pruned by objects in the tree M (Algorithm 4). To enable faster processing, our strategy is to eliminate useless paths quickly (through group level reasoning) and prioritize promising paths while checking for domination. The traversal proceeds in a depth-first style, where nodes are popped from the stack (line 5) and replaced by their children (lines 8-11). We also associate with each stack entry, a flag, *FoundCloser*. Consider an internal node, n at level n_l which has fixed the values $[u_1, u_2, \ldots, u_{n_l}]$ for the first n_l attributes. Now, the *FoundCloser* flag would be set to the following condition:

$$d_1(c_1, u_1) < d_1(c_1, v_1(Q)) \vee \ldots d_{n_l}(c_{n_l}, u_{n_l}) < d_{n_l}(c_{n_l}, v_{n_l}(Q))$$

Thus, the *FoundCloser* flag, for an internal node, records whether a value that is *strictly* closer to the corresponding value in c than that in Q, has been encountered yet.

Alg. 5 ***Prune(*** $e = [e_1, \ldots, e_m]$***, tree*** M***)***

1. $Stack = \phi$
2. $Root.FoundCloser = false$
3. $Stack.push(Root)$
4. $while(Stack \neq \phi)$
5. $\quad s = Stack.pop()$
6. $\quad if(s.isLeaf() \wedge s.FoundCloser)$
7. $\quad\quad remove\ s\ from\ the\ tree$
8. $\quad \forall p \in s.children$
9. $\quad\quad if(d_{p.level}(p,e) \leq d_{p.level}(p,q))$
10. $\quad\quad\quad p.FoundCloser = s.FoundCloser$
$\quad\quad\quad\quad\quad\quad \vee(d_{p.level}(p,e) < d_{p.level}(p,q))$
11. $\quad\quad Stack.push(p)$
12. $return\ M$

Early elimination of useless paths is achieved by pushing only such children to the stack that do not have an attribute that is more dissimilar to c than the query value for the attribute (line 9). Among the qualifying children, those that have more descendants are more promising, since the probability of finding a pruner among them is correspondingly higher. Thus, we push children on the stack in the increasing order of number of descendants (line 8), so that the ones with more descendants gets processed earlier (LIFO order). Before pushing a child on to the stack, the $FoundCloser$ flag is set accordingly (line 10) using the additional information available from the newly considered attribute. Upon encountering a leaf node to process that has its $FoundCloser$ flag set (lines 6-7), we can be sure that it can prune c since it has a value that is strictly closer to c than the query is (since the $FoundCloser$ flag is set) and it doesnt have any attribute that is farther away than the query is (since it was pushed into the stack in a previous step).

Prune(e,M): The $Prune(.,.)$ function (Algorithm 5), although similar to Algorithm 4, performs a very different task; that of removing all objects from M that can be pruned by e. The traversal is in depth first manner; however, we do not enforce any ordering while pushing nodes to the stack. *This is because we intend to prune all nodes that are prunable by e, unlike in Algorithm 4 where our objective was to find one pruner as soon as possible.* The $FoundCloser$ flag, however, has a different semantics. For an internal node n at level n_l taking values $[u_1, \ldots, u_{n_l}]$, the $FoundCloser$ represents the condition:

$$d_1(u_1, e_1) < d_1(u_1, v_1(Q)) \vee \ldots d_{n_l}(u_{n_l}, e_{n_l}) < d_{n_l}(u_{n_l}, v_{n_l}(Q))$$

Corresponding modifications are illustrated in lines 9 and 10. Upon encountering a leaf node with its $FoundCloser$ set, we would remove it from the tree (as against returning from the function as in Algorithm 4).

TRS on the Running Example: Now, we analyze the gains achieved by the TRS approach on our running example, for query $Q = [MSW, Intel, DB2]$. Having built the first batch tree in Figure 2, we now would like to check whether each of the first batch objects can be pruned by others in the tree. Consider checking for pruners of O_1 $[MSW, AMD, DB2]$; lines 4-8 would process $R1$, 1, 2, 3 and 4, in that order. The distance check in line 10 happens as long as $FoundCloser$ is false; thus, it happens only for nodes 1 and 2, since it is set to false upon considering 2. 3 would then get popped first, and since the conditions on line 6 are satisfied, the $IsPruned(.,.)$ procedure would return *true. Thus, the checking for pruners of [MSW,AMD,DB2] involves three distance checks, one each when nodes* 1, 2 *and* 4 *are pushed on to the stack.* While checking for pruners of $[MSW, Intel, DB2]$ (O_6), lines 4-8 would push just R and 1 into the stack. Note that nodes 4 and 5 would have been removed from the tree prior to calling $IsPrunable$ (line 5 of Algorithm 3).Similar to the earlier case, we incur 1 distance check while pushing 1. When 1 is popped and processed, we check whether its child 2 could be pushed to the stack; however the condition at line 9 fails in this case, and 2 is not added to the stack. *Thus, the check for pruners of [MSW,Intel,DB2] incurs* 2 *checks, as against* 4 *checks required in SRS (see Table 3).* In our second phase, our tree M is built out of the objects $\{O_3, O_6\}$. The tree is useful to save on checks when multiple objects share parts of their paths (from the root) in the tree. However, since the paths for these two objects are distinct in the tree construction (Ref. Figure 2), the $Prune(.,.)$ method makes as many checks as SRS. We enumerate the checks made, in Table 3. The number of checks made by TRS is seen to be 21% lesser than those made by SRS. We will see that the savings is more pronounced in larger datasets in Section 5.

ID	TRS			SRS		
	Ph 1	Ph 2	Total	Ph 1	Ph 2	Total
O_1	3	4	7	3	4	7
O_4	3	4	7	3	4	7
O_6	2	3	5	4	3	7
O_2	1	3	4	3	3	6
O_5	1	3	4	3	3	6
O_3	2	1	3	4	1	5
Total	**30**			**38**		

Table 3: Performance on Running Example (# Checks)

5. EXPERIMENTAL EVALUATION

We now describe our experimental study where we compare the proposed RS algorithms on real and synthetic data.

5.1 Experimental Setup

Our experiments were run on an IBM X Series with Windows Server 2003 having a Pentium 3.4 GHz processor with 2.0GB RAM. We compare the algorithms based on disk access (IO) costs, computational costs and response times. *IO Costs* are measured in terms of page IOs. Random IO is costlier than sequential IO; we plot these separately and consistently use a page size of 32KB in our experiments. The *computational time* is the sole indicator of the cost when the database can be held in memory. To isolate the computational costs from the IO costs, we use a scenario where all the objects are loaded in memory; all costs become purely computational (since IO is eliminated) then. The *response time*, which indicates the total time for a query, is the most significant measure, being the measure visible to the user. We measure this as the running time of a program where all the disk writes and reads are performed as necessary, by writing and reading from files on disk.

AL-Tree requires an ordering of attributes. Arranging the attributes in the increasing order of number of distinct values would enable better group level reasoning due to larger sized groups towards the root. Top-k query processing benefits from such an ordering [10]; we use the same strategy here.

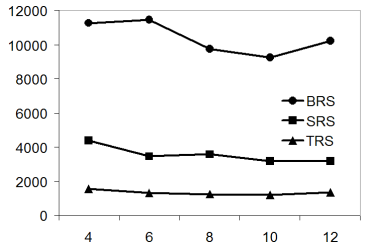

Figure 3: Computation (ms) vs. % Memory (CI)

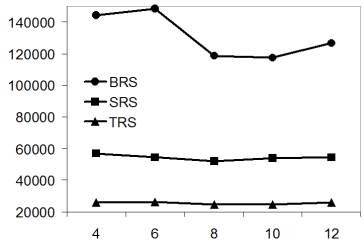

Figure 4: Computation (ms) vs. % Memory (FC)

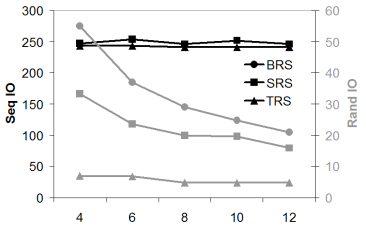

Figure 5: IO Cost vs. % Memory (CI)

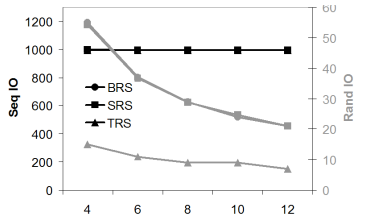

Figure 6: IO Cost vs. % Memory (FC)

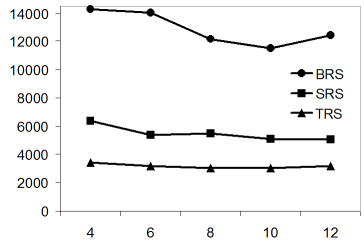

Figure 7: Resp Time (ms) vs. % Memory (CI)

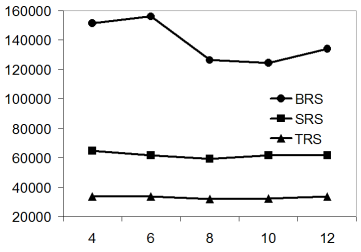

Figure 8: Resp Time (ms) vs. % Memory (FC)

5.2 Datasets

The *ForestCover* dataset[3] 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 (there were as many as 44 binary attributes among the 55 attributes present) leading to a very low data density of 0.04%. The *Census-Income* dataset[4] is a denser containing census data for 199523 people 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%. The similarity between different values of attributes are chosen randomly from the interval $[0-1]$.

Usage of synthetic data allows us to vary dataset parameters such as number of distinct values per attribute, number of data points etc. We generate synthetic data from a normal distribution, since normal distributions are said to characterize real data. Normal data is characterized by a high density around the *middle* values; this makes it tricky to generate non-metric space data since there is no global ordering of values. However, we assume an ordering of values for each attribute, and generate data to ensure that the distribution is normal and hence is concentrated around the middle values in the chosen ordering. We still generate similarities between values randomly; hence values around the middle of the chosen ordering are not designed to have high similarities to each other. We use a uniform random number generator and rejection sampling[5]. We choose the variance to be 3, and the mean to be the index of the middle attribute in the ordering chosen for data generation.

[3] http://archive.ics.uci.edu/ml/datasets/Covertype

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

[5] http://en.wikipedia.org/wiki/Rejection_Sampling

5.3 Performance on Real Data

This section details our analyses of the techniques on the real datasets, *Census-Income* (CI) and *ForestCover* (FC).

Computational Costs

The computational costs of the various techniques, across varying memory sizes (as percentage of dataset size), are illustrated in Figures 3 and 4 respectively. The relative trends are similar across the datasets; TRS is roughly 3 times and 6 times faster than SRS and BRS respectively. The group level reasoning and early pruning help reduce the computational costs for TRS significantly. The CI dataset is relatively dense, and small, both contributing to lesser computational costs. Thus, TRS takes just 2 seconds of CPU to complete processing. On the other hand, TRS takes upto 25 seconds on the sparser and bigger FC dataset, since sparseness makes it harder to find pruners. Extremely small memory sizes may impede loading a reasonable sample of the dataset in a batch in the first phase, thus making the first phase redundant since pruning would be ineffective within small batches. However, even with memory sizes of 4%, these effects are not visible, and computation costs remain consistent across increasing memory sizes thereon.

IO Costs

SRS is designed to have better first phase pruning, due to sorting. Thus, it is expected to have fewer first phase results, leading to lesser iterations in the second phase as compared to BRS. TRS further reduces the IO cost by virtue of using a compact structure to represent the in memory data. The AL-Tree, being a prefix tree, has only one node to represent each value of the first attribute, regardless of the number of objects that have that value. By maintaining objects as a tree, TRS is able to accommodate more objects using the same amount of available memory. Since we include objects in a batch as long as we have free memory, batch sizes in the TRS approach could potentially be higher than those in BRS

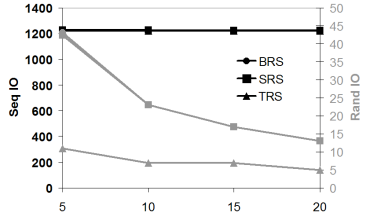

Figure 9: IO Cost (ms) vs. % Memory (Synthetic Normal Data)

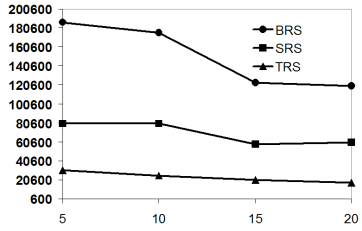

Figure 10: Resp Time (ms) vs. % Memory (Synthetic Normal Data)

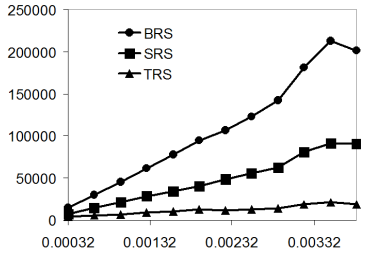

Figure 11: Computation (ms) vs. Density (Varying Dataset Size)

and SRS. This is expected to lead to fewer batches in the first phase, fewer first phase results and consequently fewer iterations in the second phase, leading to IO savings. Interestingly, in each of our experiments, the first phase pruning for all the approaches was effective enough to lead to a very small number of results to be processed in the second phase; thus, the second phase was always completed in a single pass since the intermediate results were small enough and could be loaded entire into memory. Thus, all the algorithms needed to perform just two sequential scans; consequently, sequential IO costs of all of them were found to be similar. However, the increased number of partial results in BRS and SRS as compared to TRS reflects as increased random IO costs for them. We present the IO cost charts for CI and FC datasets in Figures 5 and 6 respectively. In these charts, the sequential access costs (black lines) are plotted on the left Y-axis whereas the random accesses incurred (grey lines) are plotted on the right Y-axis. The random IO costs are seen to reduce with increasing memory sizes due to larger batch sizes and consequent reduction in intermediate results.

It may also be worth noting that IO costs contribute upto 65% of the total response time for queries on the CI dataset, whereas it is much lesser for the FC dataset. This increased contribution of IO costs to the total running time gets more pronounced with increasing density of datasets. This is expected since increased density improves the chances of finding pruners in the same block, resulting in better pruning and thus significantly reduced computational costs. In the very sparse synthetic datasets that we experiment with, in the next section, the total running times will be seen to be dominated largely by the computational expenses.

Response Times

The Reverse Skyline algorithms work by doing efficient pairwise comparisons; the pairwise comparison approach makes them quadratic in complexity. Thus, unlike linear algorithms for other similarity search operators [10, 21], the response time is dominated by the computational costs. Hence, the relative margins and absolute response times remain similar to those observed for computational costs, as seen in Figures 7 and 8 for CI and FC respectively. The TRS approach is seen to respond many times faster than the other approaches on varying memory sizes and datasets confirming it is the algorithm of choice for most scenarios.

5.4 Performance on Synthetic Data

We now compare the the approaches on synthetic normal data of varying characteristics. Unless otherwise mentioned, the available memory size is kept at 10% of the dataset size.

Varying Memory Sizes

Similar to our studies on real-world datasets, we study the trends with varying memory sizes on a dataset of 1 million objects and 5 attributes, with 50 values per attribute; we vary the available memory from 5% to 20%. The IO trends (Figure 9) are very similar to those observed for the real datasets in Figures 5 and 6. Relative behavior of the techniques in terms of response times as shown in Figure 10 is also similar to those observed for the real datasets.

Varying Dataset Sizes

For this experiment, we vary the dataset size from 0.1 to 1.2 million keeping the number of attributes and number of values per attribute constant at 5 and 50 respectively. This varies the data density from 0.0003 to 0.003. The computational time, IO costs and response times are plotted in Figures 11, 12 (the BRS line closely follows the SRS line in certain charts making it hardly visible) and 13 respectively. Computational time, as expected, is seen to take the bulk of the response time for each of the techniques. TRS is seen to outperform BRS by upto an order of magnitude and SRS by a factor of 5 in terms of computational costs and response time. BRS and SRS are seen to follow each other closely in terms of sequential IO costs whereas TRS outperforms the others (as expected) in terms of random IOs; TRS is less expensive in terms of IO too and incurs half as much of IO costs as the other approaches on the average.

Varying Number of Values per Attribute

We now keep the dataset size constant at 1 million and and vary the number of values per attribute from 45 to 70 in steps of 5 in a 5-attribute dataset, thus varying the density from 0.0005 to 0.005. With changing number of values per attribute, the dataset itself changes, thus leading to different result sets and result set sizes. This leads to widely varying computational costs (Ref. Figure 14); however, TRS is seen to outperform BRS and SRS by factors of 6 and 3 on an average. The IO behavior (Figure 15) is similar to ones observed earlier; the gap between TRS and others on random IOs, however, is seen to be wider here. The Response Times are once again dominated by computational expenses (Ref. Figure 16) with TRS outperforming the other approaches by upto 3-6 times.

Varying Number of Attributes

TRS relies on the effectiveness of a prefix-tree based structure whose depth is the number of attributes; deeper structures enable reasoning based on smaller groups closer to the leaf, whereas shallower structures have larger subsets at each

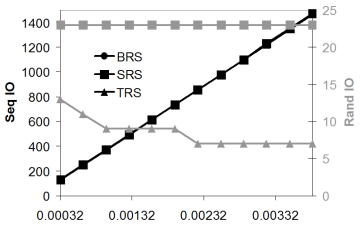

Figure 12: IO Cost vs. Density (Varying Dataset Size)

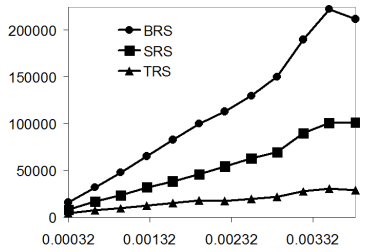

Figure 13: Resp Time (ms) vs. Density (Varying Dataset Size)

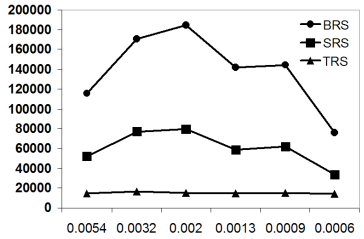

Figure 14: Computation (ms) vs. Density (Varying # values)

node. The possibility of finding pruners is significantly reduced with increased number of attributes since there are more conditions to be satisfied to enable pruning. We now analyze the performance of the techniques on a dataset size of 1 million, with 50 values per attribute, varying the number of attributes from 3 to 7 (in the process, varying the density from 8 to 1.28E-6). IO costs (Figure 17) show similar trends as in earlier experiments, with the techniques incurring similar sequential IO costs and TRS faring better in terms of random IOs. As in other cases, the response time (Figure 18, in logarithmic scale) follows the computational expense trends and TRS responds upto 5 times faster than SRS and upto 8 times faster than BRS, on the average. This shows that the incremental gains of group level reasoning and early pruning enabled by TRS scales well with the number of attributes.

5.5 Pre-processing Costs

SRS and TRS both have an additional pre-processing step of sorting the database. The sorting is independent of the query and is an one-time operation. External sorting[6] algorithms sort huge databases that cannot be loaded entirely into memory by loading parts of the database to perform an in-memory sort, and then merging the partial results efficiently. External Sorting algorithms of today can sort GBs of data within a few minutes[7]. We used the SmallText toolkit[8] to gauge the pre-processing costs over the datasets that we used in our experiments. With the memory set to 10% of the dataset size, the SmallText external sorting procedure took 3.2 seconds to sort the *ForestCover* dataset and 2.1 seconds to sort the *Census-Income* dataset. The synthetic normal 5-attribute dataset of 1 million objects took 4.2 seconds to get sorted. Thus, the pre-processing costs are seen to be negligible, for all practical settings.

5.6 Alternative Data Orderings and Querying on Attribute Subsets

In the SRS and TRS methods, the objects are sorted so that objects that share attribute values are clustered close to each other. We use a simple multi-dimensional sort based on an attribute ordering. When the number of values per attribute vary widely across attributes, the multi-dimensional sort with attributes that take lesser number of values at the top enables good group level reasoning. Additionally, the multi-dimensional sort has been found to be beneficial in top-k[10] and skyline query processing[21].

In certain cases, users may want to perform Reverse Skyline queries on subsets of attributes; for example, among the many attributes of hotels, a user may be interested in only the price and proximity to the beach. The TRS and SRS approaches can be trivially adapted to consider only the chosen subset of attributes while processing Reverse Skyline queries. However, the chosen sort order may not match the order according to the subsets of the attributes chosen. For example, consider 5 attributes and the ordering $[A_1, A_2, A_3, A_4, A_5]$ for multi-dimensional sort in SRS and TRS. The subset of attributes $\{A_1, A_2, A_3\}$ matches this order, whereas the subset $\{A_3, A_4, A_5\}$ does not. Resorting the objects based on the chosen subset of attributes at query time is very expensive and not a feasible option. Among two subsets of attributes $\{A_1, A_2, A_3\}$ and $\{A_3, A_4, A_5\}$ that could be chosen by the user, SRS/TRS would perform better on the first set since the chosen attributes occur at the top of the ordering in the multi-dimensional sort. In the second case, the chosen sort order is not favorable since objects having the same values for $\{A_3, A_4, A_5\}$ are not clustered together.In summary, the SRS/TRS approaches are expected to suffer when the chosen attributes do not form a prefix of the chosen attribute ordering. It may be noted that the ordering is more critical to the SRS approach. For example, if an object and its pruner are located 100 objects away, O(100) comparisons are required in SRS since the search for a pruner radiates away from the object in consideration until a pruner is found. On the other hand, TRS would need only as many comparisons as the number of attributes as long as an object and its pruner are in the same block.

To address this issue, we need to cluster the objects in a way that is fair to all the dimensions. Multi-dimensional tiling [11] has been found to be effective in affecting such multi-dimensional clustering of objects. Tiles are hyper-rectangles in the multi-dimensional space, formed by dividing the range of attribute values along each dimension. We explore the utility of tiling as an alternative to multi-dimensional sort, in our approaches for Reverse Skyline search. The objects within a tile are sorted as before and the tiles are ordered using a Z-order[9]. Such an ordering is expected to be less sensitive to the exact choice of the subset of attributes under consideration. We now compare the different orderings (multi-dimensional sort and tile based ordering) and the block and tree approaches that use them. We refer to the block based and tree-based approaches on the tile based ordering as $T\text{-}SRS$ and $T\text{-}TRS$ respectively. The analogous approaches on data ordered according to the multi-dimensional sort are simply the SRS and TRS ap-

[6] http://en.wikipedia.org/wiki/External_sorting

[7] http://brie.di.unipi.it/smalltext/utils.html

[8] http://brie.di.unipi.it/smalltext/

[9] http://en.wikipedia.org/wiki/Z-order_(curve)

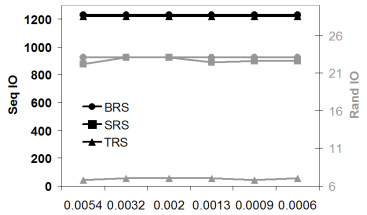

Figure 15: IO Cost vs. Density (Varying # values)

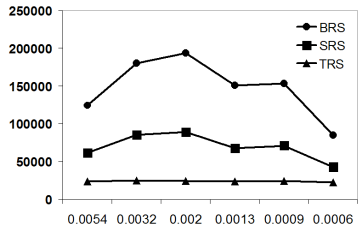

Figure 16: Resp Time (ms) vs. Density (Varying # values)

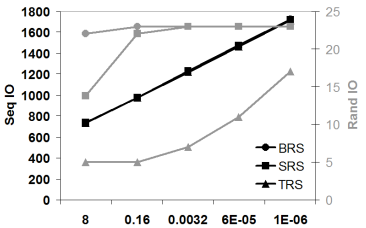

Figure 17: IO Cost vs. Density (Varying # attrs)

proaches. Response times on a dataset with $100k$ objects with 7 attributes each (50 values per attribute) are plotted against various selections of attribute subsets (for the query), in Figure 19. For the SRS method, the sort order used is $[A_1, A_2, A_3, A_4, A_5, A_6, A_7]$. As expected, SRS is seen to deteriorate when attributes that are at the top of the sorting order are omitted. The analogous approach, T-SRS is seen to be more insensitive to the attribute ordering. Similar is the case with T-TRS where the response times do not vary much across attribute selections. However, more interestingly, the TRS is able to give similar response times as T-TRS and even outperforms it when the first attribute in the sorted order is among the selected attributes. In summary, for querying on attribute subsets, tiling is effective for the SRS method, whereas the simple multi-dimensional sort is good enough for the TRS method. It also confirms the effectiveness of TRS across varying selections of attribute subsets to perform the Reverse Skyline Search on.

5.7 Applicability of Metric Space Approaches

TRS (as well as BRS and SRS) is a multi-pass algorithm, incurring one pass in the first phase and as many passes as there are intermediate result batches, in the second. However, empirically, the intermediate result sets are seen to be always small enough to be loaded into memory in one batch, thus necessitating just one pass in the second phase. Reverse skyline result sets are often small (often of cardinality $\tilde{1}$0-100, similar to observations in [9]) and intermediate results were found to be ranging only upto 4-5 times of that; thus each of our experiments needed just two passes in total.

We now consider the applicability of metric space approaches in our setting. Once a query is specified, the objects in the database can now be thought of as being in a euclidean space with the query object at the origin, and the co-ordinates being represented by the distance from the query object, on the respective dimension. Usage of metric space approaches on this Euclidean space (that is formed at query-time) requires construction of metric space indexes (e.g., R-tree) at query time. Recent R-tree construction approaches [16] work by reading the database in batches, creating batch-wise R-trees and merging them with global disk-resident R-tree. At the minimum, this involves reading the database once, and writing out as much information as twice of the database size (the data & index); thus incurs IO costs equivalent to three sequential scans of the database even in the hypothetical case where R-tree creation involves only sequential accesses. In practice, updating the disk-resident R-tree with another would require many random accesses, since the access pattern is not inherently sequential. This expensive query-time step of creating R-trees rules out the applicability of metric space approaches in our setting.

6. HANDLING NUMERIC ATTRIBUTES

If all the attributes are numeric (and thus metric), existing techniques for Reverse Skyline retrieval [9] can be used. The TRS algorithm presented in this paper is useful for categorical attributes that have non-metric dissimilarity measures. However, many real-world datasets have a combination of numeric and categorical attributes; thus it would be beneficial if TRS could handle numeric attributes as well. Group level reasoning, as enabled by TRS, becomes useful when many objects take the same value for an attribute. Such cases are common in the case of categorical attributes whose values come from a finite domain. However, this rarely holds in the case of continuous spaces (infinite domains) like numeric attributes, thus diminishing the effectiveness of reasoning on groups. We now briefly outline how discretization is an effective method to leverage group-level reasoning within the Tree-RS framework.

First Phase: Let the lower and upper bounds of the buckets to which attribute i of an object o is mapped to be denoted by $o_i.l$ and $o_i.u$ respectively, post-discretization. Consider checking whether an object c could have pruners under an internal node p (as in Algorithm 4). The condition in line 9 (and similar conditions) could now be replaced by:

$$max\{d_i(c_i.l, p_i.u), d_i(c_i.u, p_i.l)\} \leq min\{d_i(c_i.l, q_i.u), d_i(c_i.u, q_i.l)\}$$

This checks whether the maximum dissimilarity between buckets to which $v_i(c)$ and $v_i(p)$ are mapped to is less than the minimum dissimilarity between the buckets of $v_i(c)$ and $v_i(q)$. This, obviously, is stronger than a check on the dissimilarities between the actual values. Thus, there could be more false positives among first phase results; these are refined in the second phase.

Second Phase: In order to ensure that each object that could be pruned by e be removed from M, we maintain, at each leaf node, the actual numeric values assumed by each object that maps to the leaf node. Upon reaching a leaf, all such entires of objects that could be pruned by e are evicted from the leaf by means of exact checks on the numeric attributes, thus completing the operation.

7. CONCLUSIONS

In this paper, we proposed the first techniques for efficient Reverse Skyline retrieval under arbitrary non-metric similarity measures. We present various optimizations and propose techniques that use such optimizations to speed up Reverse Skyline processing. Block-based accesses are helpful in drastically reducing the number of disk scans and thus IO cost. A pre-processing step of sorting the database is seen to be helpful in keeping objects closer to their pruners,

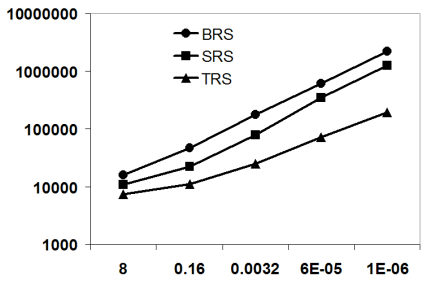

Figure 18: Resp Time (ms) vs. Density (Varying # attrs)

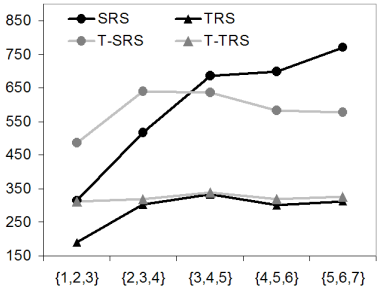

Figure 19: Resp Time (ms) vs. Attribute Subsets.

thus enabling faster pruning, eventually leading to 3-5 times speedup in Reverse Skyline Processing. Our main contribution, that of leveraging group level reasoning and early pruning to micro-optimize pruning checks, is seen to further optimize the processing to speed-up processing by upto 3-5 factors. Extensive empirical analysis over real and synthetic datasets confirm these results in many different situations. Further, TRS is seen to be consistently better than the other approaches in terms of IO costs, computational costs and response times separately, and thus is the algorithm of choice for virtually all possible scenarios.

8. REFERENCES

[1] E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *SIGMOD Conference*, pages 515–526, 2006.

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

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 1975.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[5] G.-H. Cha. Non-metric similarity ranking for image retrieval. In *DEXA*, pages 853–862, 2006.

[6] H. Chen, R. Shi, K. Furuse, and N. Ohbo. Finding rknn straightforwardly with large secondary storage. In *INGS*, 2008.

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

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, 1997.

[9] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, pages 291–302, 2007.

[10] P. M. Deshpande, Deepak. P, and K. Kummamuru. Efficient online top-k retrieval with arbitrary similarity measures. In *EDBT*, pages 356–367, 2008.

[11] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *SIGMOD*, 1998.

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

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

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[15] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD*, 2000.

[16] T. Lee, B. Moon, and S. Lee. Bulk insertion for r-trees by seeded clustering. *Data Knowl. Eng.*, 59:86–106, October 2006.

[17] X. Lian and L. C. 0002. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD*, 2008.

[18] X. Lian and L. C. 0002. Reverse skyline search in uncertain databases. *ACM Trans. Database Syst.*, 35(1), 2010.

[19] G. Murphy and D. Medin. The role of theories in conceptual coherence. In *Psychological Review*, 1985.

[20] Deepak. P and P. Deshpande. Efficient rknn retrieval with arbitrary non-metric similarity measures. *PVLDB*, 3(1), 2010.

[21] Deepak. P, P. M. Deshpande, D. Majumdar, and R. Krishnapuram. Efficient skyline retrieval with arbitrary similarity measures. In *EDBT*, 2009.

[22] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD Conference*, pages 467–478, 2003.

[23] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High dimensional reverse nearest neighbor queries. In *CIKM*, 2003.

[24] T. Skopal and J. Lokoc. Nm-tree: Flexible approximate similarity search in metric and non-metric spaces. In *DEXA*, pages 312–325, 2008.

[25] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *In SIGMOD Workshop on DMKD*, pages 44–53, 2000.

[26] M. Vlachos, D. Gunopulos, and G. Kollios. Robust similarity measures for mobile object trajectories. In *DEXA 2002*, 2002.

[27] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *ICDE*, 2001.

[28] J. L. Yanmin Luo, Canhong Lian and H. Chen. Finding rknn by compressed straightforward index. In *ISKE*, 2008.

[29] L. Zhu, C. Li, and H. Chen. Efficient computation of reverse skyline on data stream. In *CSO*, 2009.