

A Study on External Memory Scan-Based Skyline Algorithms

Nikos Bikakis^{1,2}, Dimitris Sacharidis², and Timos Sellis³

¹ National Technical University of Athens, Greece

² IMIS, “Athena” R.C., Greece

³ RMIT University, Australia

Abstract. Skyline queries return the set of non-dominated tuples, where a tuple is dominated if there exists another with better values on all attributes. In the past few years the problem has been studied extensively, and a great number of external memory algorithms have been proposed. We thoroughly study the most important scan-based methods, which perform a number of passes over the database in order to extract the skyline. Although these algorithms are specifically designed to operate in external memory, there are many implementation details which are neglected, as well as several design choices resulting in different flavors for these basic methods. We perform an extensive experimental evaluation using real and synthetic data. We conclude that specific design choices can have a significant impact on performance. We also demonstrate that, contrary to common belief, simpler skyline algorithm can be much faster than methods based on pre-processing.

Keywords: Experimental evaluation, experimental survey, disk-based algorithm.

1 Introduction

The *skyline query*, or skyline operator as it was introduced in [3], has in the past few years received great attention in the data management community. Given a database of objects, the skyline query returns those objects which are not dominated. An object *dominates* another, if it has better values on all attributes, and strictly better value on at least one. Finding the skyline is also known as the Pareto-optimal set, or maximal vectors problem in multi-objective optimization research, where it has been studied extensively in the past, but only for in-memory computations. For example the well-known divide and conquer algorithm of [7] has complexity $O(N \log^{d-2} N)$, for $d \geq 2$, where N is the number of objects, and d their dimensionality; the algorithm is optimal for $d = 3$.

The interest in external memory algorithms has sparked after the seminal work in [3]. The most efficient method in terms of worst-case Input/Output (I/O) operations is the algorithm in [15], which requires in the worst case $O\left(\left(N/B\right) \log_{M/B}^{d-2} \left(N/B\right)\right)$ I/Os, where M is the memory size and B the block (minimum unit of transfer in an I/O operation) size in terms of objects. However,

in practice, other external-memory algorithms proposed over the past years can be faster.

This work studies in detail an important class of practical algorithms, the *scan-based skyline algorithms*. An algorithm of this class performs multiple passes over an input file, where the input file in the first pass is the database, and in a subsequent pass it is the output of the previous pass. The algorithm terminates when the output file remains empty after a pass concludes. Generally speaking, during each pass, the algorithm maintains in main memory a small *window* of incomparable objects, which it uses to remove dominated objects from the input file. Any object not dominated is written to the output file.

Although the studied algorithms are specifically designed to operate in external memory, little attention has been given to important implementation details regarding memory management. For example, all algorithms assume that the unit of transfer during an I/O operation is the object, whereas in a real system is the block, i.e., a set of objects. Our work addresses such shortcomings by introducing a more realistic I/O model that better captures performance in a real system. Furthermore, by thoroughly studying the core computational challenge in these algorithms, which is the management of the objects within the window, we introduce several novel potentially interesting policies.

Summarizing, the contributions of our study are the following:

- Based on a standard external memory model [1], we appropriately adapt four popular scan-based algorithms, addressing in detail neglected implementation details regarding memory management.
- We focus on the core processing of scan-based algorithms, the management of objects maintained in the in-memory window. In particular, we introduce various policies for two tasks: traversing the window and evicting objects from the window. Both tasks can have significant consequences in the number of required I/Os and in the CPU time.
- We experimentally evaluate concrete disk-based implementations, rather than simulations, of all studied algorithms and derive useful conclusions for synthetic and real datasets. In particular, we demonstrate that, in many cases and contrary to common belief, algorithms that pre-process (typically, sort) the database are not faster.
- We perform an extensive study of our proposed policies, and reach the conclusion that in some settings (dimensionality and dataset distribution) these policies can reduce the number of dominance checks by more than 50%.

2 Preliminaries

2.1 Definitions

Let \mathcal{O} be a set of *d-dimensional objects*. Each object $o \in \mathcal{O}$ is represented by its *attributes* $o = (o^1, o^2, \dots, o^d)$. The *domain* of each attribute, is the positive real numbers set \mathbb{R}^+ . Without loss of generality, we assume that an object o_1 is *better* than another object o_2 on an attribute j , iff $o_1^j < o_2^j$. An object o_1

dominates another object o_2 , denoted by $o_1 \succ o_2$, iff (1) $\forall i \in [1, d], o_1^i \leq o_2^i$ and (2) $\exists j \in [1, d], o_1^j < o_2^j$. The *skyline* of an object set \mathcal{O} , denoted as $SL(\mathcal{O})$, is the set of objects in \mathcal{O} that are not dominated by any other object of \mathcal{O} . Formally, $SL(\mathcal{O}) = \{o_i \in \mathcal{O} \mid \nexists o_k \in \mathcal{O} : o_k \succ o_i\}$.

2.2 External Memory I/O Model

This section describes an external memory model, similar to that of [1]. The unit of transfer between the main memory and the external memory (i.e., the disk) is a single *block*.¹ Any external memory algorithm, like the skyline methods, read/write blocks from/to disk *files*. We assume that files are stored contiguously on disk, and therefore a new block is written always at the end of a file.

We denote as $N = |\mathcal{O}|$ the size of the database, i.e., N is the total number of objects to be processed. We measure the fixed size B of a block in terms of objects (tuples). Similarly, main memory can fit M objects, with the requirements that $M < N$ (and often much smaller) to justify the need for external memory algorithms, and $M > 2B$ to support basic in-memory operations.

We next discuss Input/Output (I/O) operations. We assume no input or output buffers, so that blocks from the disk are transferred directly to (resp. from) the disk from (resp. to) the main memory. Equivalently, the input/output buffers share the same memory of size M with the algorithm.

We categorize I/O operations in two ways. Naturally, a *read* transfers data from the disk, whereas a *write* transfers data to the disk. The second categorization is based on the number of blocks that are transferred. Note that a read (resp. write) operation transfers at least one block and at most $\lfloor \frac{M}{B} \rfloor$ blocks into main memory (resp. disk). We also remark that in disks, the *seek time*, i.e., the time it takes for the head to reach the exact position on the ever spinning disk where data is to be read or written, is a crucial parameter in disk performance. Reading or writing k consecutive blocks on the disk is much faster than reading or writing k blocks in arbitrary positions on the disk. The reason is that only one seek is required in the first case, compared to the k seeks for the second. Therefore, we distinguish between *sequential* and *random* I/Os. A random I/O incorporates the seek time, whereas a sequential I/O does not. For example, when a procedure reads k blocks sequentially from the disk, we say that it incurs 1 random read and $k - 1$ sequential reads.

3 A Model for Scan-Based Skyline Algorithms

3.1 Design Choices

All skyline algorithms maintain a set of objects, termed *window*, which consists of possible skyline objects, actual skyline objects, or some arbitrary objects in general. A common procedure found in all algorithms is the following. Given some candidate object not in the window, *traverse* the window and determine if

¹ [1] assumes that P blocks can be transferred concurrently; in this work we set $P = 1$.

the candidate object is dominated by a window object, and, if not, additionally determine the window objects that it dominates. Upon completion of the traversal and if the candidate is not dominated, the skyline algorithm may choose to insert it into the window, possibly *evicting* some window objects.

In the aforementioned general procedure, we identify and focus on two distinct design choices. The first is the *traversal policy* that determines the order in which window objects are considered and thus dominance checks are made. This design choice directly affects the number of dominance checks performed and thus the running time of the algorithm. An ideal (but unrealistic) traversal policy would require only one dominance check in the case that the candidate is dominated, i.e., visit only a dominating window object, and/or visit only those window objects which the candidate dominates.

The second design choice is the *eviction policy* that determines which window object(s) to remove so as to make room for the candidate object. This choice essentially determines the dominance power of the window, and can thus indirectly influence both the number of future dominance checks and the number of future I/O operations.

We define four window traversal policies. The *sequential traversal policy* (*sqT*), where window objects are traversed *sequentially*, i.e., in the order they are stored. This policy is the one adopted by all existing algorithms. The *random traversal policy* (*rdT*), where window objects are traversed in *random* order. This policy is used to gauge the effect of others. The *entropy-based traversal policy* (*enT*), where window objects are traversed in ascending order of their *entropy* (i.e., $\sum_{i=1}^d \ln(o^i + 1)$) values. Intuitively, an object with a low entropy value has greater dominance potential as it dominates a large volume of the space.

In addition to these traversal policies, we define various ranking schemes for objects, which will be discussed later. These schemes attempt to capture the dominance potential of an object, with higher ranks suggesting greater potential. Particularly, we consider the following traversal policies. The *ranked-based traversal policy* (*rkT*), where window objects are traversed in descending order based on their *rank* values. Moreover, we consider three hybrid random-, rank-based traversal policies. The *highest-random traversal policy* (*hgRdT*), where the k objects with the highest rank are traversed first, in descending order of their rank; then, the random traversal policy is adopted. The *lowest-random traversal policy* (*lwRdT*), where the k objects with the lowest rank are compared first, before continuing with a random traversal. Finally, the *recent-random traversal policy* (*rcRdT*), where the k most recently read objects are compared first, before continuing with a random traversal.

Moreover, we define three eviction policies. The *append eviction policy* (*apE*), where the *last* inserted object is removed. This is the policy adopted by the majority of existing algorithms. The *entropy-based eviction policy* (*enE*), where the object with the *highest entropy* value is removed. Finally, the *ranked-based eviction policy* (*rkE*), where the object with the *lowest rank* value is removed. In case of ties in entropy or rank values, the most recent object is evicted.

We next discuss ranking schemes used in the ranked-based traversal and eviction policies. Each window object is assigned a rank value, initially set to zero. Intuitively, the rank serves to identify “promising” objects with high dominance power, i.e., objects that dominate a great number of other objects. Then, the skyline algorithm can exploit this information in order to reduce the required dominance checks by starting the window traversal from promising objects, and/or evict non-promising objects.

We define three ranking schemes. *r0R*: the rank of an object o at a time instance t , is equal to the number of objects that have been dominated by o until t . In other words, this ranking scheme counts the number of objects dominated by o . *r1R*: this ranking is similar to *r0R*. However, it also considers the number of objects that have been dominated by the objects that o dominates. Let $rank(o)$ denote the rank of an object o . Assume that object o_1 dominates o_2 . Then, the rank of o_1 after dominating o_2 is equal to $rank(o_1) + rank(o_2) + 1$. *r2R*: this ranking assigns two values for each object o , its *r1R* value, as well as the number of times o is compared with another object and none of them is dominated (i.e., the number of incomparable dominance checks). The *r1R* value is primarily considered to rank window objects, while the number of incomparable check is only considered to solve ties; the more incomparable checks an object has, the lower its rank.

3.2 Algorithm Adaptations for the I/O Model

BNL. The *Block Nested Loop* (BNL) [3] algorithm is one of the first external memory algorithms for skyline computation. All computations in BNL occur during the window traversal. Therefore, BNL uses a window as big as the memory allows. In particular, let W denote the number of objects stored in the window, and let O_b denote the number of objects scheduled for writing to disk (i.e., in the output buffer). The remaining memory of size $I_b = M - W - O_b$ serves as the input buffer, to retrieve objects from the disk. Note that the size of the I/O buffers I_b and O_b vary during the execution of BNL, subject to the restriction that the size of the input buffer is always at least one disk block, i.e., $I_b \geq B$, and that the output buffer never exceeds a disk block, i.e., $O_b \leq B$; we discuss later how BNL enforces this requirements.

We next describe memory management in the BNL algorithm. BNL performs a number of passes, where in each an input file is read. For the first pass, the input file is the database, whereas the input file in subsequent passes is created at the previous pass. BNL terminates when the input file is empty. During a pass, the input file is read in *chunks*, i.e., sets of blocks. In particular, each read operation transfers into main memory exactly $\lfloor \frac{I_b}{B} \rfloor$ blocks from disk, incurring thus 1 random and $\lfloor \frac{I_b}{B} \rfloor - 1$ sequential I/Os. On the other hand, whenever the output buffer fills, i.e., $O_b = B$, a write operation transfers into disk exactly 1 block and incurs 1 random I/O.

We now discuss what happens when a chunk of objects is transferred into the input buffer within the main memory. For each object o in the input buffer, BNL

traverses the window, adopting the sequential traversal policy (*sqT*). Then, BNL performs a two-way dominance check between o and a window object w . If o is dominated by w , o is discarded and the traversal stops. Otherwise, if o dominates w , object w is simply removed from the window.

At the end of the traversal, if o has not been discarded, it is appended in the window. If W becomes greater than $M - O_b - B$, BNL needs to move an object from the window to the output buffer to make sure that enough space exists for the input buffer. In particular, BNL applies the append eviction policy (*apE*), and selects the last inserted object, which is o , to move into the output buffer. If after this eviction, the output buffer contains $O_b = B$ objects, its contents are written to the file, which will become the input file of the next pass.

A final issue is how BNL identifies an object o to be a skyline object, BNL must make sure that o is dominance checked with all surviving objects in the input file. When this can be guaranteed, o is removed from the window and returned as a result. This process is implemented through a *timestamp mechanism*; details can be found in [3].

SFS. The *Sort Filter Skyline* (SFS) [4] algorithm is similar to BNL with one significant exception: the database is first sorted by an external sort procedure according to a monotonic scoring function. SFS can use any function defined in Section 3.1.

Similar to BNL, the SFS algorithm employs the sequential window traversal policy (*sqT*) and the append eviction policy (*apE*). There exist, however, two differences with respect to BNL. Due to the sorting, dominance checks during window traversal are one-way. That is an object o is only checked for dominance by a window object w . In addition, the skyline identification in SFS is simpler than BNL. At the end of each pass, all window objects are guaranteed to be results and are thus removed and returned.

LESS. The *Linear Elimination Sort for Skyline* (LESS) [5] algorithm improves on the basic idea of SFS, by performing dominance checks during the external sort procedure. Recall that standard external sort performs a number of passes over the input data. The so-called zero pass (or sort pass) brings into main memory M objects, sorts them in-memory and writes them to disk. Then, the k -th (merge) pass of external sort, reads into main memory blocks from up to $\lfloor M/B \rfloor - 1$ files created in the previous pass, merges the objects and writes the result to disk.

LESS changes the external sort procedure in two ways. First, during the zero pass, LESS maintains a window of size W_0 objects as an elimination filter to prune objects during sorting. Thus the remaining memory $M - W_0$ is used for the in-memory sorting. The window is initially populated after reading the first $M - W_0$ objects by selecting those with the lowest entropy scores. Then for each object o read from the disk and before sorting them in-memory, LESS performs a window traversal. In particular, LESS employs the sequential traversal policy (*sqT*) performing a one-way dominance check, i.e., it only checks if o is dominated. Upon comparing all input objects with the window, the object with

the lowest entropy o_h is identified. Then, another sequential window traversal (sqT) begins, this time checking if o_h dominates the objects in the window. If o_h survives, it is appended in the window, evicting the object with the highest entropy score, i.e., the entropy-based eviction policy (enE) is enforced.

The second change in the external sort procedure is during its last pass, where LESS maintains a window of size W objects. In this pass, as well as any subsequent skyline processing passes, LESS operates exactly like SFS. That is the sequential traversal policy (sqT) is used, one-way dominance checks are made, and window objects are removed according to the append eviction policy (epE).

RAND. In the Randomized multi-pass streaming (RAND) algorithm [13], each pass in RAND consists of three phases, where each scans the input file of the previous pass. Therefore, each pass essentially corresponds to three reads of the input file. In the first phase, the input file is read and a window of maximum size $W = M - B$ is populated with randomly sampled input objects (using reservoir sampling).

In the second phase, the input file is again read one block at a time, while the window of W objects remain in memory. For each input object o , the algorithm traverses the window in sequential order (sqT), performing one-way dominance checks. If a window object w is dominated by o , w is replaced by o . Note that, at the end of this phase, all window objects are skyline objects, and can be returned. However, they are not removed from memory.

In the third phase, for each input object o , RAND performs another sequential traversal of the window (sqT), this time performing an inverse one-way dominance check. If o is dominated by a window object w , or if o and w correspond to the same object, RAND discards o . Otherwise it is written on a file on the disk, serving as the input file for the next pass. At the end of this phase, the memory is cleaned.

4 Related Work

External memory skyline algorithms can be classified into three categories: (1) scan-based, (2) index-based, and (3) partitioning-based algorithms.

The *scan-based* approaches perform multiple passes over the dataset and use a small window of candidate objects, which is used to prune dominated objects. The algorithms of this category can be further classified into two approaches: with and without pre-processing. Algorithms of the first category, directly process the set of objects, in the order in which they are stored, or produced (e.g., in the case of pipelining multiple operators). The BNL [3] and RAND [13] algorithms, detailed in Section 3.1, lie in this category. On the other hand, methods in the second category perform an external sort of the objects before, or parallel to the skyline computation. The SFS [4] and LESS [5], also detailed in Section 3.1, belong to this category. Other algorithm, include Sort and Limit Skyline algorithm (SaLSa) [2], which is similar to SFS and additionally introduces a condition for early terminating the input file scan, and Skyline Operator

on Anti-correlated Distributions (SOAD) [14], which is also similar to SFS but uses different sorting functions for different sets of attributes.

In *index-based* approaches, various types of indices are used to guide the search for skyline points and prune large parts of the space. The most well-known and efficient method is the Branch and Bound Skyline (BBS) [12] algorithm. BBS employs an R-tree, and is shown to be I/O optimal with respect to this index. Similarly, the Nearest Neighbor algorithm (NN) [6] also uses an R-tree performing multiple nearest neighbor searches to identify skyline objects. A bitmap structure is used by Bitmap [16] algorithm to encode the input data. In the Index [16] algorithm, several B-trees are used to index the data, one per dimension. Other methods, e.g., [9,10], employ a space-filling curve, such as the Z-order curve, and use a single-dimensional index. The Lattice Skyline (LS) algorithm [11] builds a specialized data structure for low-cardinality domains.

In the *partitioning-based* approaches, algorithms divide the initial space into several partitions. The first algorithm in this category, D&C [3] computes the skyline objects adopting the divide-and-conquer paradigm. A similar approach with stronger theoretical guarantees is presented in [15]. Recently, partitioning-based skyline algorithms which also consider the notion of incomparability are proposed in [17,8]. OSP [17] attempts to reduce the number of checks between incomparable points by recursively partition the skyline points. BSkyTree [8] enhances [17] by considering both the notions of dominance and incomparability while partitioning the space.

5 Experimental Analysis

5.1 Setting

Datasets. Our experimental evaluation involves both synthetic and real datasets. To construct synthetic datasets, we consider the three standard distribution types broadly used in the skyline literature. In particular, the distributions are: anti-correlated (ANT), correlated (CORR), and independent (IND). The synthetic datasets are created using the generator developed by the authors of [3].

We also perform experiments on three real datasets. *NBA* dataset consists of 17,264 objects, containing statistics of basketball players. For each player we consider 5 statistics (i.e., points, rebound, assist, steal blocks). *House* is 6-dimensional dataset consists of 127,931 objects. Each object, represents the money spent in one year by an American family for six different types of expenditures (e.g., gas, electricity, water, heating, etc.). Finally, *Colour* is a 9-dimensional dataset, which contains 68,040 objects, representing the first three moments of the RGB color distribution of an image.

Implementation. All algorithms, described in Section 3.1, were written in C++, compiled with gcc, and experiments were performed on a 2.6GHz CPU. In order to accurately convey the effect of I/O operations, we disable the operating system caching, and perform direct and synchronous I/O's.

The size of each object is set equal to 100 bytes, as was the case in the experimental evaluation of the works that introduced the algorithms under investigation. Finally, the size of block is set to 2048 bytes; hence each block contains 20 object.

Metrics. To gauge efficiency of all algorithms, we measure: (1) the number of disk I/O operations, which are distinguished into four categories, read, write operations, performed during the pre-processing phase (i.e., sorting) if any, and read, write operations performed during the main computational phase; (2) the number of dominance checks; (3) the time spent solely on CPU processing denoted as CPU Time and measured in seconds; (4) the total execution time, denoted as Total Time and measured in seconds; In all cases the reported time values are the averages of 5 executions.

5.2 Algorithms Comparison

Table 1 lists the parameters and the range of values examined. In each experiment, we vary a single parameter and set the remaining to their default

Table 1. Parameters

Description	Parameter	Values
Number of Objects	N	50k, 100K, 500K , 1M, 5M
Number of Attributes	d	3, 5 , 7, 9, 15
Memory Size	$M/N(\%)$	0.15%, 0.5% 1% , 5%, 10%

(bold) values. SFS and LESS sort according to the entropy function. During pass zero in LESS, the window is set to one block.

Varying the Number of Objects. In this experiment, we vary the number of objects from 50K up to 5M and measure the total time, number of I/O's and dominance checks, and CPU time, in Figures 1–4.

The important conclusions from Figure 1 are two. First, RAND and BNL outperform the other methods in anti-correlated datasets. This is explained as follows. Note that the CPU time mainly captures the time spent for the following task: dominance checks, data sorting in case of LESS/SFS, and skyline identification, in case of BNL. From Figure 4 we can conclude that BNL spends a lot of CPU time in skyline identification. BNL requires the same or more CPU time than RAND, while BNL performs fewer dominance checks than RAND. This is more clear in the case of independent and correlated datasets where the cost for dominance checks is lower compared to the anti-correlated dataset. In these datasets, the BNL CPU time increased sharply as the cardinality increases.

The second conclusion is that, in independent and correlated datasets, the performance of BNL quickly degrades as the cardinality increases. This is due to the increase of the window size, which in turn makes window maintenance and skyline identification more difficult.

Figure 2 shows the I/O operations performed by the algorithms. We observe that BNL outperforms the other methods in almost all settings. Particularly, in the correlated dataset, LESS is very close to BNL. Also, we can observe that, in general, the percentage of write operations in LESS and SFS is much higher than in BNL and RAND. We should remark that, the write operations are generally

more expensive compared to the read operations. Finally, for LESS and SFS, we can observe that the larger amount of I/O operations are performed during the sorting phase.

Regarding the number of dominance checks, shown in Figure 3, LESS and SFS perform the fewest, while RAND the most, in all cases. Figure 4 shows the CPU time spent by the methods. SFS spends more CPU time than LESS even though they perform similar number of dominance checks; this is because SFS sorts a larger number of object than LESS. Finally, as previously mentioned, BNL spends considerable CPU time for skyline identification.

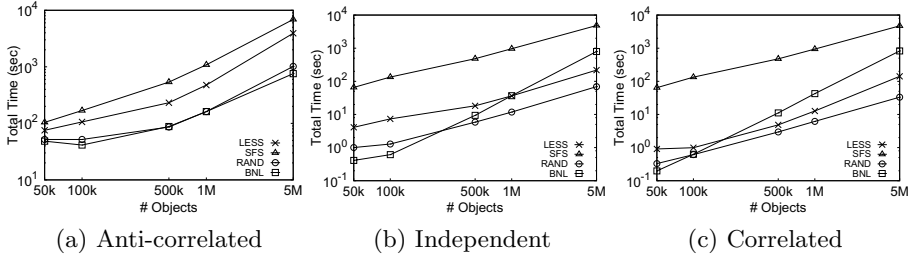


Fig. 1. Total Time: Varying Number of Objects

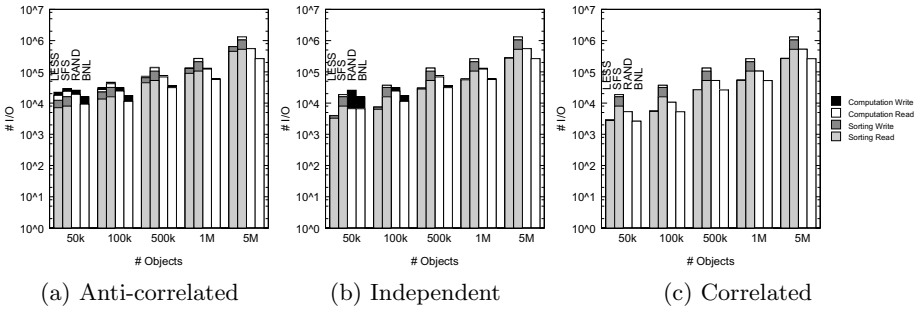


Fig. 2. I/O Operations: Varying Number of Objects

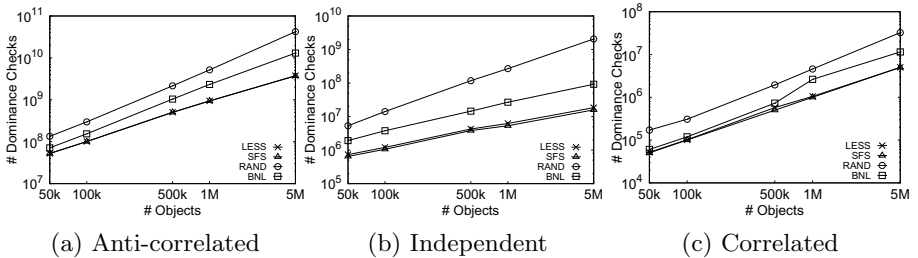


Fig. 3. Dominance Checks: Varying Number of Objects

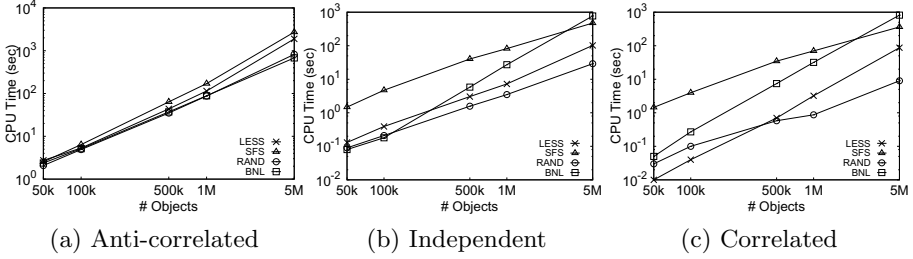


Fig. 4. CPU Time: Varying Number of Objects

Varying the Number of Dimensions. In this experiment we investigate the performance as we vary the number of dimensions from 3 up to 15. In Figure 5 where the total time is depicted, the performance of all methods become almost the same for anti-correlated and independent datasets, as the dimensionality increases. In the correlated dataset, the skyline can fit in main memory, hence BNL and RAND require only a few passes, while SFS and LESS waste time sorting the data.

Regarding I/O’s (Figure 6), BNL outperforms all other methods in all cases, while LESS is the second best method. Similarly, as in Figure 2, LESS and SFS performs noticeable more write operations compared to BNL and RAND. Figure 7 shows that LESS and SFS outperforms the other method, performing the same number of dominance checks. Finally, CPU time is presented in Figure 8, where once again the cost for skyline identification is noticeable for BNL.

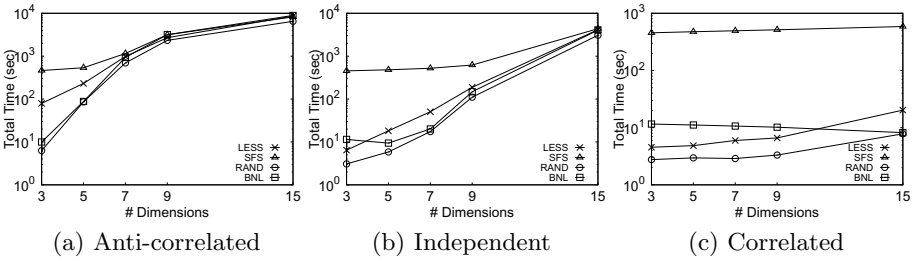


Fig. 5. Total Time: Varying Number of Attributes

Varying the Memory Size. In Figure 9, we vary the size of the available memory. In general, the total time here, follows the trend of I/O operations. We observe that the required time of all methods decreased sharply for memory sizes up to 1%. However, beyond this point, the time is almost stable as the memory size increases, with the exception of BNL, where the time slightly increases (due to the skyline identification cost w.r.t. window size).

Table 2. Real Datasets: Total Time (sec)

Dataset	LESS	SFS	RAND	BNL
House	30.11	178.21	15.25	4.98
Colour	14.43	90.73	3.70	1.28
NBA	9.45	26.68	0.71	0.41

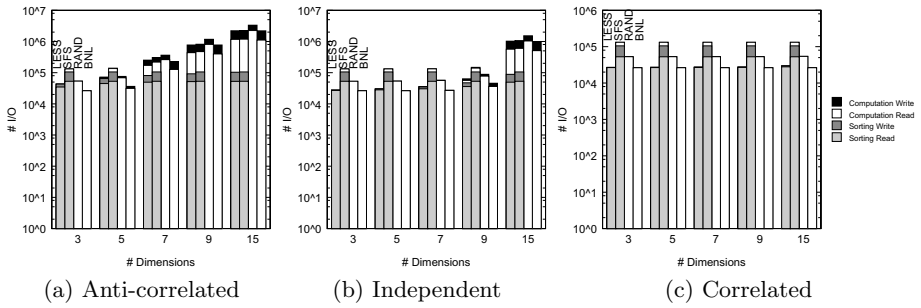


Fig. 6. I/O Operations: Varying Number of Attributes

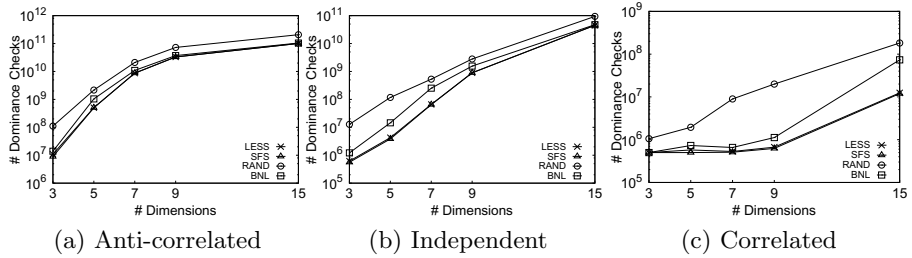


Fig. 7. Dominance Checks: Varying Number of Attributes

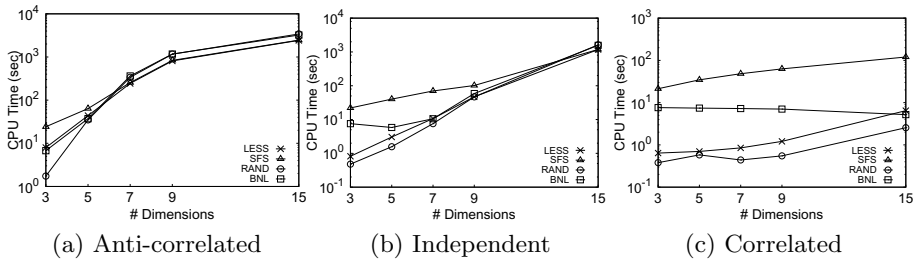


Fig. 8. CPU Time: Varying Number of Attributes

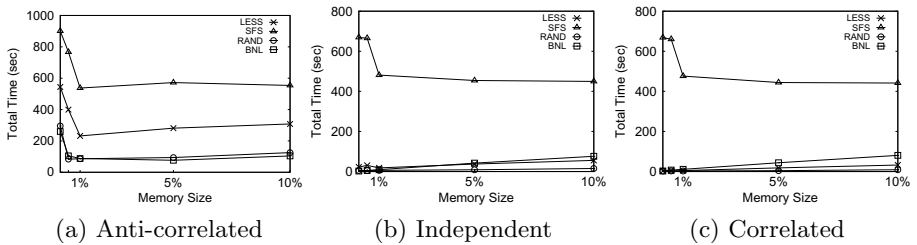


Fig. 9. Total Time: Varying Memory Size

Real Datasets. In this experiment, we evaluate our methods using the real datasets described in Section 5.1. Table 2 summarizes the results, presenting the total time required by all methods. We observe that BNL outperforms the other methods in all datasets in terms of total time. RAND outperforms the other methods in all cases, while SFS is the worst. Note that, in House and Colour datasets, RAND performs more dominance checks, and more I/O operations, than LESS. However, LESS requires more total time, due to larger number of write operations, and the CPU time spend for sorting.

5.3 Policies Evaluation

In this experiment, we study the effect of different window policies in scan-based skyline algorithms. Particularly, we use BNL and SFS algorithms and we employ several traversal and eviction and policies, in conjunction with different ranking schemes. The effect of policies in LESS are similar to those in SFS and are not shown. Regarding RAND, only the window traversal policy affects its performance; its effect is not dramatic and hence it is also not shown.

All results are presented w.r.t. the original algorithms. That is, let m be a measurement for the original algorithm, and m' be the corresponding measurement for an examined variation. In this case, the measurement presented for the variation is $1 + (m' - m)/m$.

BNL. We first study the performance of BNL under the 10 most important policy and ranking scheme combinations. Figure 10 shows the I/O operations performed by the BNL flavors. As we can see, none of the examined variations performs significant better than the original algorithm. In almost all cases, the I/O performance of most variations is very close to the original. The reason is that the append eviction policy (apE), adopted by the original BNL already performs very well for two reasons. First, the apE policy always removes objects that have not dominated any other object. This way, the policy indirectly implements a dominance-oriented criterion. Second, the apE policy always removes the most recently read object, which is important for BNL. A just read object, requires the most time (compared to other objects in the window) in order to be identified as a skyline, thus propagated to the results and freeing memory. Hence, by keeping “older” objects we increase the probability of freeing memory in the near future. Still it is possible to marginally decrease the number of I/Os.

Figure 11 shows the number of dominance checks performed. We can observe that, in several cases, the variants that adopt rank-based traversal, perform significant fewer dominance checks than the original. Particularly, the rkT/rkE/r1R and rkT/rkE/r2R variants outperform the others in almost all cases, in independent and correlated datasets, by up to 50%. Similar results also hold for low dimensionalities in the anti-correlated dataset. However, this does not hold in more dimensions, due to the explosion of skyline objects in anti-correlated datasets.

SFS. Here, as in the previous experiment, we examine the performance of SFS algorithm adopting several policies. Similar to BNL, none of SFS variants

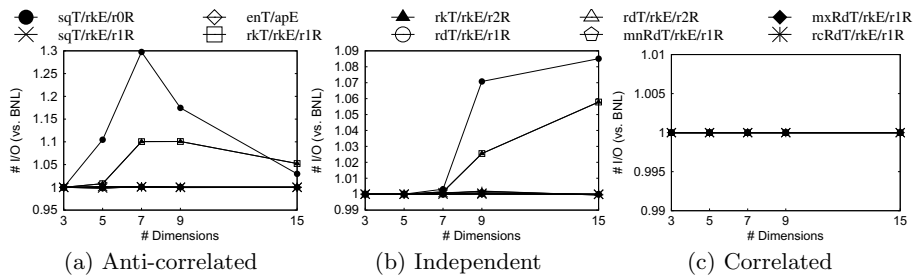


Fig. 10. BNL Policies (I/O Operations): Varying Number of Attributes

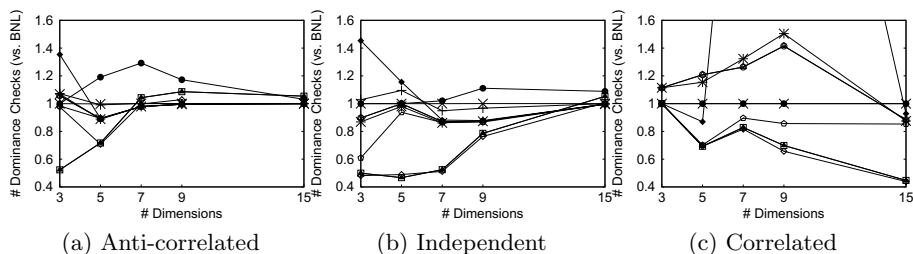


Fig. 11. BNL Policies (Dominance Checks): Varying Number of Attributes

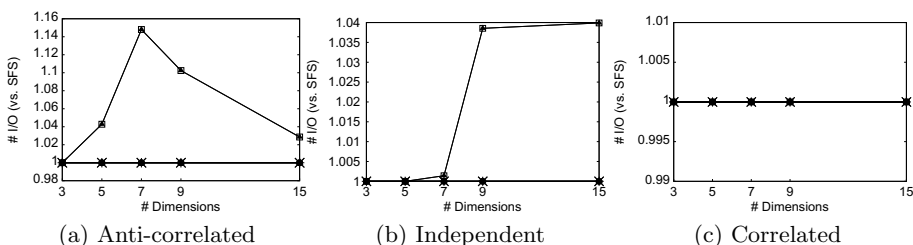


Fig. 12. SFS Policies (I/O Operations): Varying Number of Attributes

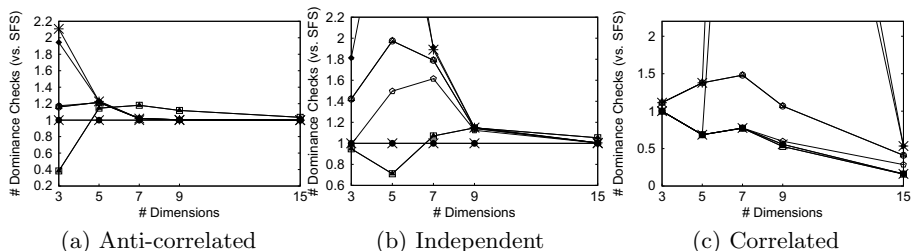


Fig. 13. SFS Policies (Dominance Checks): Varying Number of Attributes

perform noticeable fewer I/O operations (Figure 12). Regarding the dominance checks (Figure 13), in anti-correlated and independent datasets, most of variants have similar performance to the original algorithm. Only for correlated datasets, ranked-based policies exhibit significant performance gains.

5.4 Discussion

In an I/O-sensitive setting, i.e., when I/O operations cost significantly more than CPU cycles, BNL seems to be the ideal choice, as it performs less I/O operations than all other methods in almost all settings. Additionally, BNL and RAND perform less write operation than the other methods. On the other hand, in a CPU-sensitive setting, LESS and RAND seem to be good choices. LESS performs the fewest dominance checks, while RAND doesn't spend time for sorting the data, or for skyline identification. Finally, regarding the policies tested, the rank-based ones show significant gains but only in CPU-sensitive settings.

Acknowledgements. This research has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program “Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) – Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. *Commun. ACM* 31(9) (1988)
2. Bartolini, I., Ciaccia, P., Patella, M.: Efficient sort-based skyline evaluation. *TODS* 33(4) (2008)
3. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: *ICDE* (2001)
4. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: *ICDE* (2003)
5. Godfrey, P., Shipley, R., Gryz, J.: Algorithms and analyses for maximal vector computation. *VLDBJ* 16(1) (2007)
6. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: An online algorithm for skyline queries. In: *VLDB* (2002)
7. Kung, H.T., Luccio, F., Preparata, F.P.: On finding the maxima of a set of vectors. *Journal of the ACM* 22(4) (1975)
8. Lee, J., Hwang, S.W.: Bskytrees: scalable skyline computation using a balanced pivot selection. In: *EDBT* (2010)
9. Lee, K.C.K., Zheng, B., Li, H., Lee, W.C.: Approaching the skyline in z order. In: *VLDB* (2007)
10. Liu, B., Chan, C.Y.: Zinc: Efficient indexing for skyline computation. *VLDB* 4(3) (2010)
11. Morse, M.D., Patel, J.M., Jagadish, H.V.: Efficient skyline computation over low-cardinality domains. In: *VLDB* (2007)
12. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. *TODS* 30(1) (2005)
13. Sarma, A.D., Lall, A., Nanongkai, D., Xu, J.: Randomized multi-pass streaming skyline algorithms. *VLDB* 2(1) (2009)
14. Shang, H., Kitsuregawa, M.: Skyline operator on anti-correlated distributions, vol. 6 (2013)
15. Sheng, C., Tao, Y.: Worst-case i/o-efficient skyline algorithms. *TODS* 37(4) (2012)
16. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient progressive skyline computation. In: *VLDB* (2001)
17. Zhang, S., Mamoulis, N., Cheung, D.W.: Scalable skyline computation using object-based space partitioning. In: *SIGMOD* (2009)