Volume 4, Issue 7, July 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Skyline Computation Algorithms - A Study

Apeksha Aggarwal, Harsh Kumar Verma
Department of Computer Science and Engineering
Dr B.R. Ambedkar NIT, Jalandhar, India

Abstract— A skyline query operator is designed to find the set of interesting data points (objects) over a large dimensional data collection, satisfying a set of possibly contradicting conditions. In this paper, we provide an in-depth coverage of skyline algorithms. The paper consists of three parts. First, skyline operator with one of its application is discussed. Second, we represent history of skyline algorithms developed till date along with their pros and cons. Finally, we present the efficiency of skyline query processing algorithms on the basis of time complexities over high dimensional large datasets.

Keywords— Skyline, Multi-criteria Decision Making, User defined Preferences, Sorting-Based Algorithm, Performance.

I. INTRODUCTION

Suppose we are going to a holiday destination lets say a beach and we are looking for a hotel that is cheap and close to the beach. Unfortunately, these two goals are complementary as the hotels near the beach tend to be more expensive. The database system in this case is unable to decide which hotel is best for us, but it can at least present us all *interesting* hotels. Interesting are all hotels that are not worse than any other hotel in both dimensions. We call this set of interesting hotels the *Skyline*[1]. From the Skyline, we can now make our final decision, thereby weighing our personal preferences for price and distance to the beach.

Computing the Skyline is known as the maximum vector problem [2]. We use the term Skyline because of its graphical representation (see below).

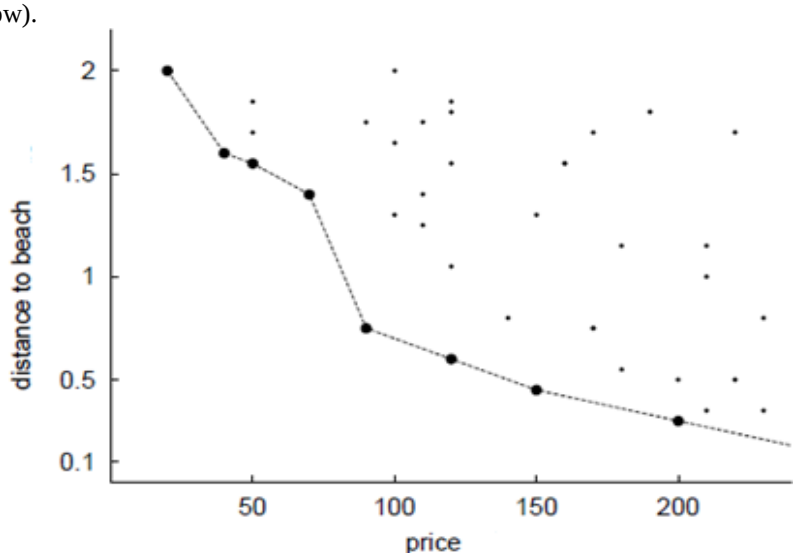

Fig 1. Skyline of Hotels

More formally, the Skyline is defined as the set of those points which are not dominated by any other point. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension. For example, a hotel with *price* = $50 and *distance* = 0.8 miles dominates a hotel with *price* = $100 and *distance* = 1.0 miles.

The big open problem with skyline queries is finding skylines over a large high dimensional dataset in real time. Many existing skyline algorithms work effectively with small datasets of low dimensionality. Their performance degrades drastically when the data size is huge and the dimensionality is high. Several recent research proposed distributed skyline query processing solutions are discussed in this work.

The rest of the paper is organized as follows:

Section II, describes the classification of various skyline based algorithms. Section III discusses sorting based algorithms in detail. Along with their advantages and disadvantages over previous work. Section IV discusses performance of various algorithms, comparing against each other under a variety of settings, along with time complexities of their expected performance. Finally, Section V concludes the paper with some directions for future work.

II. CLASSIFICATION OF SKYLINE ALGORITHMS

In this section, we provide an in-depth review of various models of skyline computations proposed to date. Classification of algorithms on the basis of class of methods they use to prune the non skyline points can be done into four types: (A) Sorting-based algorithms, (B) Hierarchical Index- based algorithms, (C) Divide-and-Conquer based algorithms, (D) Cube-based. Further Classification of these algorithms is shown in Figure 2.

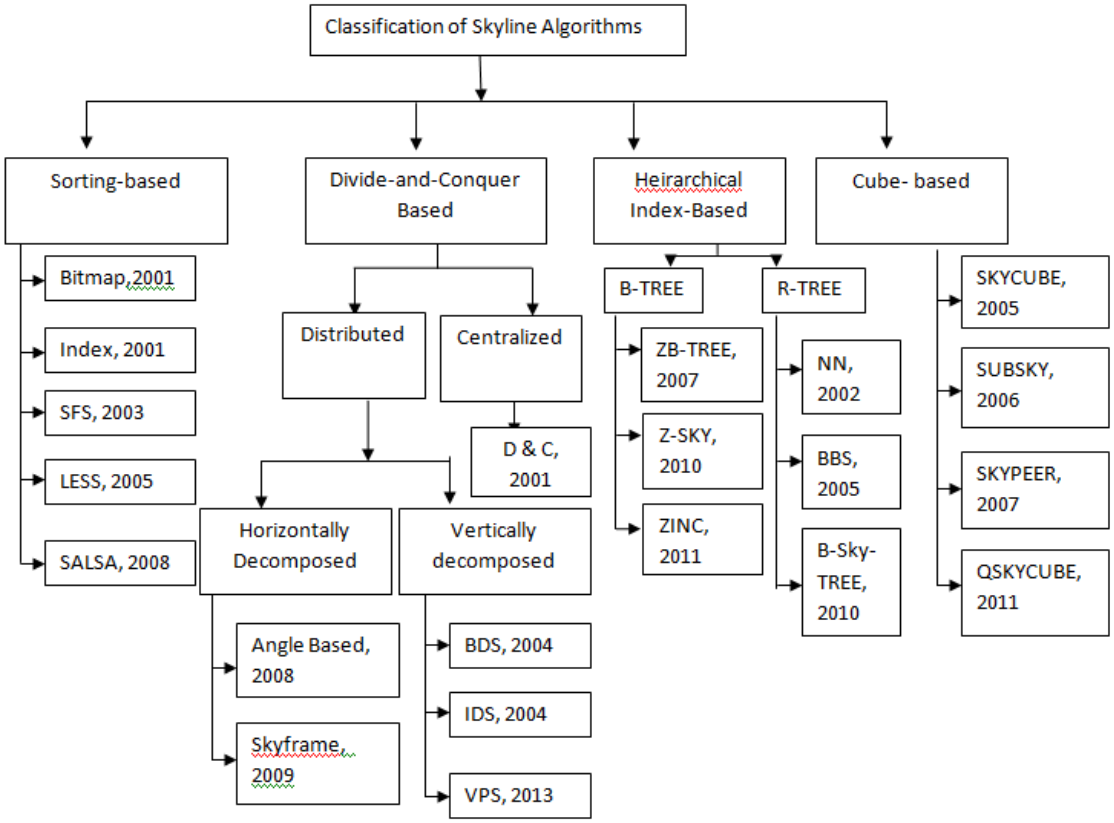

Fig 2. Classification of various skyline algorithms.

A. Sorting-based algorithms.

Sorting based algorithms like BNL[1] sort the data objects topologically according to decreasing/increasing scores of a monotone function, the computation of the skyline becomes straightforward because objects below a certain threshold cannot be a part of the skyline. Hence, the sorting based approaches can efficiently reduce the number of candidate objects and thus save the computing cost. A key factor affecting the performance of the sorting based algorithms is the choice of the sorting function and the threshold. This approach, however, requires high-cost sorting process to prune non-skyline objects. Some of the sorting-based algorithms are described in section III.

B. Hierarchical Index-based algorithms.

The rationale of hierarchical index-based algorithms is two folds: First, the use of popular index structures, such as B-tree and R-tree[3] presents a straightforward way to drastically reduce the size of a skyline candidate set. Second, the data points that are nearer the origin point have higher chance of being the skyline. Thus, the computation of the skyline can be implemented through *k* nearest neighbor search (*k*NN) [4] . The advantage of hierarchical index-based approach is its progressive behavior that can quickly return the initial results without having to scan the entire dataset. However, its applicability is limited by the necessity to index a given dataset prior to compute the skyline. This approach also has some other inherent setbacks, limiting their usefulness to only some cases. This approach adopts sophisticated techniques such as the smart partitions of B-tree index, the clever use of R-tree and the intelligent use of multicore architectures [10] to accelerate the skyline computation by parallelizing the most CPU-intensive parts, the dominance tests, as well as the fundamental limitation of hierarchical index based solutions.

There has been a lot of research on the skyline query computation problem, most of which are focused on data attribute domains that are *totally ordered (TO)*, where the best value for a domain is either its maximum or minimum value. However, in many applications, some of the attribute domains are *partially ordered (PO)* such as interval data (e.g. temporal intervals), type hierarchies, and set-valued domains, where two domain values can be incomparable. A number of recent research work [2-8] like BBS algorithm, has started to address the more general skyline computation problem where the data attributes can include a combination of TO and PO domains.

Recently, a new index method called ZB-tree [6] has been proposed for computing skyline queries for Total order domains which has better performance than BBS. The ZB-tree, which is an extension of the B+-tree, is based on interleaving the bitstring representations of attribute values using the Z-order to achieve a good clustering of the data records that facilitates efficient data pruning and minimizes the number of dominance comparisons.

Given the superior performance of ZB-tree[6] over BBS for Total order domains, one question that arises is whether the ZB-tree approach can outperform the state-of-the-art BBS-based TSS approach, when extended to handle Partial order domains.

So a new indexing approach, called ZINC [7] (for Z-order Indexing with Nested Codes), that combines ZB-tree with a novel *nested encoding scheme* for PO domains. While our nested encoding scheme is a general scheme that can encode any partial order, the design is targeted to optimize the encoding of commonly used partial orders for user preferences which we believe to have simple or moderately complex structures.

Kossmann, *et al.*[8] observed that the skyline problem is closely related to the nearest neighbor (NN) search problem. They proposed an algorithm that returns skyline objects progressively by applying nearest neighbor search on an R*- tree indexed dataset recursively. The current most efficient method is *Branch-and-Bound Skyline*(*BBS*), proposed by Papadias, *et al.*[11], which is a progressive algorithm based on the best-first nearest neighbor (BF-NN) algorithm. Instead of searching for nearest neighbor repeatedly, it directly prunes using the R*-tree structure.

C. Divide-and-Conquer based algorithms

The main idea of these algorithms is to recursively divide a given dataset into partitions until it fits into the memory and then it computes the global skyline by progressively merging the local skylines. The basic divide-and-conquer algorithm of [1] works as follows:

1. Compute the median *mp* (or some approximate median) of the input for some dimension *dp*. Divide the input into two partitions. *P*1 contains all tuples whose value of attribute *dp* is better than *mp*. *P*2 contains all other tuples.
2. Compute the Skylines *S*1 of *P*1 and *S*2 of *P*2. This is done by recursively applying the whole algorithm to *P*1 and *P*2 i.e., *P*1 and *P*2 are again partitioned. The recursive partitioning stops if a partition contains only one (or very few) tuples. In this case, computing the Skyline is trivial.
3. Compute the overall Skyline as the result of *merging* *S*1 and *S*2. That is, eliminate those tuples of *S*2 which are dominated by a tuple in *S*1. (None of the tuples in *S*1 can be dominated by a tuple in *S*2 because a tuple in *S*1 is better in dimension *dp* than every tuple of *S*2.)

Most challenging is Step 3. The main trick of this step is shown in Figure 3. The idea is to partition both *S*1 and *S*2 using an (approximate) median *mg* for some other dimension *dg*, with $dg \neq dp$. As a result, we obtain four partitions: *S*1,1*; S*1,2*; S*2,1*; S*2,2. *S*1,*i* is better than *S*2,*i* in dimension *dp* and *Si*,1 is better than *Si*,2 in dimension *dg* ($i = 1,2$). Now, we need to merge *S*1,1 and *S*2,1, *S*1,1 and *S*2,2, and *S*1,2 and *S*2,2.

The beauty is that we need not merge *S*1,2 and *S*2,1 because the tuples of these two sets are guaranteed to be incomparable. Merging *S*1,1 and *S*2,1 (and the other pairs) is done by recursively applying the *merge* function. That is, *S*1,1 and *S*2,1 are again partitioned. The recursion of the *merge* function terminates if all dimensions have been considered or if one of the partitions is empty or contains only one tuple; in all these cases the *merge* function is trivial. The algorithm has also been described in great detail in [1]

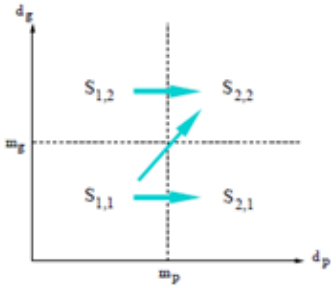

Fig 3 Basic Merge

A main disadvantage of this approach is that the average performance deteriorates as the dimensionality increases, because in higher dimensional space, the non-skyline points have higher chance to belong to the local skyline of their partition and thus increase the sizes of local skylines significantly.

The skyline operator has received considerable attention in the literature of centralized databases [3], [5], [10], [11] and distributed databases with horizontally and vertically decomposed datasets.

Horizontal decompositions [6], [13], [14], [15],where each server stores a subset of the records. On the other hand the only work on distributed skyline processing for vertically partitioned data is , which aims at minimizing the communication cost considering that the client retrieves m attribute values for a set of records from various servers. Specifically, each server 1) maintains the ID and exactly one dimension di of every record in Distributed System, 2) sorts all objects in ascending order of di at a preprocessing step, and 3) allows both sorted access (i.e., get the next record with the lowest di), or random access (i.e., given a record ID, obtain di). Balke et al. [21] propose two solutions called basic distributed skyline (BDS) and improved distributed skyline (IDS).

In BDS the client first retrieves attribute values from the servers in a round-robin manner, using sorted accesses(the next record with the lowest dimension), until it reaches an anchor point P_{anc} (anchor point that dominates, and hence eliminates, a large number of records), at all servers. Records not encountered in any of the servers are worse than Panc on every dimension, and therefore dominated by P_{anc}. Thus, the skyline is computed using only the points discovered before P_{anc}.

Assume data set of four attributes are distributed over four servers and sorted in ascending order. For BDS Anchor point A is discovered at the fifth roundrobin iteration at server N2 (let's say). At this time, the client stops the sorted accesses, obtains (using random accesses) the remaining dimensions of all records encountered before A in some server, and computes Skyline.

As opposed to BDS, which performs round-robin sorted accesses, IDS guides the search toward more promising servers (i.e., where an anchor point is likely to be found early) by interleaving sorted(get the next record with lowest dimension) and random accesses(given a record ID obtain the dimension). In 2013 VPS [21] was proposed which combines positive aspects of above 2 algorithms.

D. Cube-based algorithms.

Theoretically, there could be 2*d*-1 different skyline queries over a *d*-dimensional dataset since different users may have different preferences. Therefore, one way to improve skyline computation is to pre-compute all possible skylines in advance instead of computing each skyline at runtime on demand. However, this pre-computation of the skyline on every subspace can incur prohibitive cost. One way to manage the cost is to efficiently pre-compute the results of all possible skylines by sharing the computation of multiple related skyline queries. This approach includes the set of representative techniques for amortizing redundant computation among multiple cuboids using more sophisticated structures. SUBSKY[12] and SKYPEER[4] are 2 main cube based algorithms discussed.

III. SORTING BASED ALGORITHMS

A. Block Nested Loop (BNL)

Intuitively, a straightforward approach to compute the skyline is to compare each point *p* with every other point. If *p* is not dominated, then it is a part of the skyline. BNL builds on this concept by scanning the data file and keeping a list of candidate skyline points in main memory. The first data point is inserted into the list. For each subsequent point *p*, there are three cases:

(i) If *p* is dominated by any point in the list, it is discarded as it is not part of the skyline.
(ii) If *p* dominates any point in the list, it is inserted into the list, and all points in the list dominated by *p* are dropped.
(iii) If *p* is neither dominated, nor dominates, any point in the list, it is inserted into the list as it may be part of the Skyline.

The list is self-organizing because every point found dominating other points is moved to the top. This reduces the number of comparisons as points that dominate multiple other points are likely to be checked first. A problem of BNL is that the list may become larger than the main memory. When this happens, all points falling in third case (cases (i) and (ii) do not increase the list size), are added to a temporary file. This fact necessitates multiple passes of BNL. In particular, after the algorithm finishes scanning the data file, only points that were inserted in the list before the creation of the temporary file are guaranteed to be in the skyline and are output. The remaining points must be compared against the ones in the temporary file. Thus, BNL has to be executed again, this time using the temporary (instead of the data) file as input. The advantage of BNL is its wide applicability, since it can be used for any dimensionality without indexing or sorting the data file. Its main problems are the reliance on main memory (a small memory may lead to numerous iterations) and its inadequacy for on-line processing (it has to read the entire data file before it returns the first skyline point).

B. BITMAP

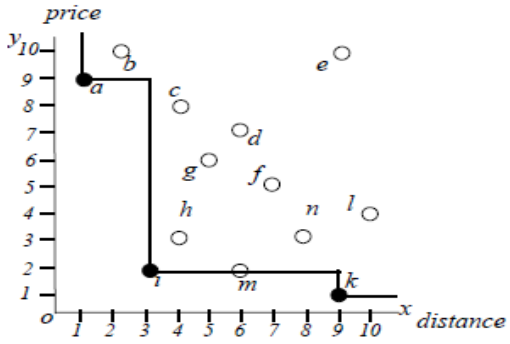

Fig 4. Example Data Set and Skyline

This technique [8F] encodes in bitmaps all the information required to decide whether a point is in the Skyline. A data point *p* =*p1*,*p2*, ..., *pd*), where *d* is the number of dimensions, is mapped to a *m*-bit vector, where *m* is the total number of distinct values over all dimensions. Let *ki* be the total number of distinct values on the *i*th dimension (i.e., *m* =Σ*i*=1~*d ki*). In Figure 4, for example, there are *k1*=*k2*=10 distinct values on the x-, y-dimensions and *m* =20. Assume that *pi* is the *ji*-th smallest number on the *i*th axis; then, it is represented by *ki* bits, where the (*ki*-*ji* +1) most significant bits are 1, and the remaining ones 0. Table 1 shows the bitmaps for points in Figure 4. Since point *a* has the smallest value (1) on the x-axis, all bits of *a1* are 1. Similarly, since *a2* (=9) is the 9-th smallest on the y-axis, the first 10-9+1=2 bits of its representation are 1, while the remaining ones are 0.

TABLE1 Bitmap Approach

id	coordinate	bitmap representation
a	(1,9)	(1111111111, 1100000000)
b	(2,10)	(1111111110, 1000000000)
c	**(4,8)**	(1111111000, 1110000000)
d	(6,7)	(1111100000, 1111000000)
e	(9,10)	(1100000000, 1000000000)
f	(7,5)	(1111000000, 1111110000)
g	(5,6)	(1111110000, 1111100000)
h	(4,3)	(1111111000, 1111111100)
i	(3,2)	(1111111100, 1111111110)
k	(9,1)	(1100000000, 1111111111)
l	(10,4)	(1000000000, 1111111000)
m	(6,2)	(1111100000, 1111111110)
n	(8,3)	(1110000000, 1111111100)

Consider now that we want to decide whether a point, e.g., *c* with bitmap representation (1111111000, 1110000000), belongs to the skyline. The most significant bits whose value is 1, are the 4th and the 8th, on dimensions *x* and *y*, respectively. The algorithm creates two bit-strings, *cX* = 1110000110000 and *cY* = 0011011111111, by juxtaposing the corresponding bits (i.e., 4th and 8th) of every point. In Table 1, these bit-strings (shown in bold) contain 13 bits (one from each object, starting from *a* and ending with *n*). The 1's in the result of *cX*&*cY*=0010000110000, indicate the points that dominate *c*, i.e., *c*, *h* and *i*. Obviously, if there is more than a single 1, the considered point is not in the skyline2. The same operations are repeated for every point in the dataset, to obtain the entire skyline. The efficiency of *bitmap* relies on the speed of bit-wise operations. The approach can quickly return the first few skyline points according to their insertion order (e.g., alphabetical order in Table 1), but cannot adapt to different user preferences, which is an important property of a good skyline algorithm. Furthermore, the computation of the entire skyline is expensive because, for each point inspected, it must retrieve the bitmaps of all points in order to obtain the juxtapositions. Also the space consumption may be prohibitive, if the number of distinct values is large. Finally, the technique is not suitable for dynamic datasets where insertions may alter the rankings of attribute values.

C. INDEX

The "index" approach[8] organizes a set of *d*-dimensional points into *d* lists such that a point $p = (p1, p2, \ldots, pd)$ is assigned to the *i*th list ($1 \le i \le d$), if and only if, its coordinate *pi* on the *i*th axis is the minimum among all dimensions, or formally: $pi \le pj$ for all $j \ne i$. Table 2 shows the lists for the dataset of Figure 4. Points in each list are sorted in ascending order of their minimum coordinate (*minC*, for short) and indexed by a B-tree. A *batch* in the *i*th list consists of points that have the same *i*th coordinate (i.e., *minC*). In Table 2, every point of list 1 constitutes an individual batch because all x-coordinates are different. Points in list 2 are divided into 5 batches {*k*}, {*i*,*m*}, {*h*,*n*}, {*l*} and {*f*}.

TABLE 2 The Index Approach

list 1		list 2	
a (1, 9)	*minC*=1	*k* (9, 1)	*minC*=1
b (2, 10)	*minC*=2	*i* (3, 2), *m* (6, 2)	*minC*=2
c (4, 8)	*minC*=4	*h* (4, 3), *n* (8, 3)	*minC*=3
g (5, 6)	*minC*=5	*l* (10, 4)	*minC*=4
d (6, 7)	*minC*=6	*f* (7, 5)	*minC*=5
e (9, 10)	*minC*=9		

Initially, the algorithm loads the first batch of each list, and handles the one with the minimum *minC*. In Table 2, the first batches {*a*}, {*k*} have identical *minC*=1, in which case the algorithm handles the batch from list 1. Processing a batch involves (i) computing the skyline inside the batch, and (ii) among the computed points, it adds the ones not dominated by any of the already-found skyline points into the skyline list. Continuing the example, since batch {*a*} contains a single point and no skyline point is found so far, *a* is added to the skyline list. The next batch {*b*} in list 1 has *minC*=2; thus, the algorithm handles batch {*k*} from list 2. Since *k* is not dominated by *a*, it is inserted in the skyline. Similarly, the next batch handled is {*b*} from list 1, where *b* is dominated by point *a* (already in the skyline). The algorithm proceeds with batch {*i*,*m*}, computes the skyline inside the batch that contains a single point *i* (i.e., *i* dominates *m*), and adds *i* to the skyline. At this step the algorithm does not need to proceed further, because both coordinates of *i* are smaller than or equal to the *minC* (i.e., 4, 3) of the next batches (i.e., {*c*}, {*h*,*n*}) of lists 1 and 2. This means that all the remaining points (in both lists) are dominated by *i* and the algorithm terminates with {*a*,*i*,*k*}.

Although this technique can quickly return skyline points at the top of the lists, it has several disadvantages. First, as with the *bitmap* approach, the order that the skyline points are returned is fixed, not supporting user-defined preferences. Second, the lists computed for *d* dimensions cannot be used to retrieve the skyline on any subset of the dimensions. In general, in order to support queries for arbitrary dimensionality subsets, an exponential number of lists must be pre-computed.

D. SFS

Our *sort-filter-skyline* algorithm[16], works as follows. It is multi-pass as is BNL, and likewise keeps a window to collect skyline tuples. The table is sorted first in some topological sort compatible with the skyline criteria. Let the sorted table be T_0 The algorithm proceeds as BNL, except now, when a tuple is added to the window during pass T_i we know that it is skyline. No tuple following it in T_i ican dominate it, by the theorem: Any *total order of the tuples of R with respect to* any *monotone scoring function (ordered from highest to lowest score) is a topological sort with respect to the skyline dominance partial relation.* Thus the tuple can be output as skyline immediately, and a copy placed in the window. Window operations in SFare less expensive, since no replacement checking is needed. SFS has the following advantages over BNL

1. There are good optimizations applicable to SFS, but not to BNL.
2. SFS is well behaved in a relational engine setting. BNL is badly behaved.SFSis guaranteed to work within the optimal number of passes, while BNL is not. SFS is not CPU-bound, as is BNL.
3. SFS provides an ordering, which is potentially useful within the query plan.
4. SFS does not block on output, so is output-pipelinable.

E. LESS

LESS (linear elimination sort for skyline)[17] that combines aspects of SFS and BNL.Thus LESS sorts the records initially, then filters the records via a skyline-filter (SF) window, as does SFS. LESS makes two major changes:

1. it uses an elimination-filter (EF) window in pass zero of the external sort routine to eliminate records quickly; and
2. it combines the final pass of the external sort with the first skyline-filter (SF) pass.

The external sort routine used to sort the records is integrated into LESS. Let b be the number of buffer pool frames allocated to LESS. Pass zero of the standard external sort routine reads in b pages of the data, sorts the records across those b pages (say, using quicksort), and writes the b sorted pages out as a b-length sorted run. All subsequent passes of external sort are merge passes. During a merge pass, external sort does a number of (b − 1)-way merges, consuming all the runs created by the previous pass. For each merge, (up to) b − 1 of the runs created by the previous pass are read in one page at a time, and written out as a single sorted run.

LESS additionally eliminates records during pass zero of its external-sort phase. It does this by maintaining a small elimination-filter window. Copies of the records with the best entropy scores seen so far are kept in the EF window.The EF window acts similarly to the elimination window used by BNL.

In effect, LESS has all of SFS's benefits with no additional disadvantages. LESS should consistently perform better than SFS. Some buffer-pool space is allocated to the EF window in pass zero for LESS which is not for SFS. Consequently, the initial runs produced by LESS's pass zero are smaller than SFS's; this may occasionally force that LESS will require an additional pass to complete the sort. Of course LESS saves a pass since it combines the last sort pass with the first skyline pass. LESS also has BNL's advantages, but effectively

none of its disadvantages. BNL has the overhead of tracking when window records can be promoted as known maximals. LESS does not need this. Maximals are identified more efficiently once the input is effectively sorted. Thus LESS has the same advantages as does SFS in comparison to BNL

F. SALSA

SaLSa (for *Sort and Limit Skyline algorithm*)[18], differs from other generic algorithms in that it consistently *limits* the number of points on which dominance tests need to be executed. The design of SaLSa is based on two key concepts: first, a sorting step of the input data and, second, the observation that, for suitably chosen sorting functions, it is indeed possible to compute the skyline by looking only at a (hopefully small) prefix of the sorted input stream. While the idea of presorting is not new, since it is at the heart of the SFS algorithm by Chomicki et al. [2003], in that algorithm it was mainly advocated as a way to bring in the first positions those points that are likely to dominate many other points, thus leading to a reduction in the number of dominance tests. On the other hand, sorting data in SaLSa is mainly used as a means to stop fetching points from the input stream. In other terms, SaLSa relies on sorting functions that can guarantee that *all* points beyond a certain level in the input stream are dominated by some already seen point, which we conveniently call the *stop point.*

IV. PERFORMANCE COMPARISON

Time Complexities of various algorithms are discussed below in Table 3.

TABLE 3 Performance Comparison of Various Skyline Algorithms.

Algorithm	Best-case	Average-case	Worst-case
Divide & Conquer (2001)	O(kn)	$\Omega(k^{.5}\,2^{2k}n)$	$O(kn^2)$
BNL (2001)	O(kn)	-	$O(kn^2)$
SFS (2003)	O(nlgn + kn)	O(nlgn + kn)	$O(kn^2)$
LESS (2006)	O(kn)	O(kn)	$O(kn^2)$
SaLSa(2008)	O(f(n))+ O(1)	O(f(n))+ O(1)	O(f(n))+ O(1)

Where O(f(n)) is the complexity of chosing sorting function. Examples of various Sorting functions is discussed in [18].

V. CONCLUSION

All existing database algorithms for skyline computation have several deficiencies, which severely limit their applicability.*BNL* is progressive i.e. first results are reported instantly. But *BNL* and *D&C* are very sensitive to main memory size and the dataset characteristics. Furthermore *D &* C is fast but is not progressive. *Bitmap* is applicable only for datasets with small attribute domains and cannot efficiently handle updates. *Bitmap* is also not progressive. *Index* a new approach is progressive but does not support user-defined preferences and cannot be used for skyline queries on a subset of the dimensions.

LESS has all of SFS's benefits with no additional disadvantages. LESS should consistently perform better than SFS. LESS also has BNL's advantages, but effectively none of its disadvantages. LESS, which improves over the existing skyline algorithms, and we prove that its average-case performance is O(kn).

SaLSa algorithm, whose innovative feature is the ability of computing the result without having to apply dominance tests to all the objects (points) in the input relation. This is achieved by presorting the data using a monotone *limiting* function, and then checking that unread data are all dominated by a so-called *stop point*.

SaLSa is indeed effective in reducing the number of points to be read, thus also particularly attractive when the skyline logic runs on a client with a limited bandwidth connection.

Finally, we want to explore new variations of skyline queries, in addition to the ones proposed in Section III. Current trends in the area of skyline computation and the further research directions in the same is highlighted in [20].

REFERENCES

[1] S. Börzsönyi, D. Kossmann and K. Stocker. "The skyline operator". *Proceedings of ICDE*, (2001), 421-430.

[2] P. Godfrey, R. Shipley, and J. Gryz, "Algorithms and Analyses for Maximal Vector Computation," Int'l J. Very Large Data Bases,vol. 16, no. 1, 5-28 2007.

[3] Theodoridis, Y., Stefanakis, E., Sellis, "T. Efficient Cost Models for Spatial Queries Using R-trees". TKDE, 12(1):19-32, 2000.

[4] Kossmann, D., Ramsak, F., Rost, S. "Shooting Stars in the Sky: an Online Algorithm for Skyline Queries". VLDB, 2002.

[5] Dimitris Papadias et al., "An Optimal and Progressive Algorithm for Skyline Queries". *ACM SIGMOD'2003*, June 9-12, San Diego, California, USA..

[6] Ken C. K. Lee et al. " Approaching Skyline in Z-order" VLDB, Sept 07.

[7] Bin Liu et al., "ZINC Efficient Indexing for Skyline Computation", Proceedings of the VLDB Endowment, Vol. 4, No. 3 August 29th - September 3rd 2011.

[8] Kossmann, D., Ramsak, F., Rost, S. Shooting Stars in the Sky: an Online Algorithm for Skyline Queries. VLDB, 2002.

[9] Yunjun Gao, et al. "On efficient reverse skyline query processing" ELSEVIER, Expert Systems with Applications 41 (2014) 3237–3249.

[10] K. Lee, B. Zhang, H. Li, and W.-C. Lee, "Approaching the Skyline in Z Order," Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB), 2007.

[11] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive Skyline Computation in Database Systems," ACM Trans. Database Systems, vol. 30, no. 1, pp. 41-82, 2005.

[12] Y. Tao, X. Xiao, and J. Pei, "SUBSKY: Efficient Computation of Skylines in Subspaces," Proc. 22nd Int'l Conf. Data Eng. (ICDE),2006.

[13] A. Vlachou, C. Doulkeridis, and Y. Kotidis, "Angle-Based Space Partitioning for Efficient Parallel Skyline Computation," Proc. ACM SIGMOD Int'l Conf. Management of Data, 2008.

[14] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis, "SKYPEER: Efficient Subspace Skyline Computation over Distributed Data," Proc. Int'l Conf. Data Eng. (ICDE), 2007.

[15] S. Wang, Q.H. Vu, B.C. Ooi, A.K.H. Tung, and L. Xu, "Skyframe: A Framework for Skyline Query Processing in Peer-to-Peer Systems," Int'l J. Conf. Very Large Data Bases, vol. 18, no. 1,pp. 345-362, 2009.

[16] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang, "Skyline with PresortingTechnical Report , Computer Science, York University, Toronto, ON, Canada, Oct. , 2005.

[17] Godfrey, Shipley, & Gryz, "Algorithms and Analyses for Maximal Vector Computation". VLDB Journal 2006 p.1 of 22, August 2006.

[18] ILARIA BARTOLINI, PAOLO CIACCIA, and MARCO PATELLA "Efficient Sort-Based Skyline Evaluation". ACM Transactions on Database Systems, Vol. 33, No. 4, Article 31, Publication date: November 2008.

[19] Su Min Jang and Choon Seo Park "Skyline Minimum Vector". 12th International Asia-Pacific Web Conference, 2010.

[20] Ms. R.D. Kulkarni1 and Prof. Dr. B.F. Momin, " Future Research Directions in Skyline Computation". International Journal of Computer Engineering Science (IJCES) , Volume 2 Issue 5 (May 2012).

[21] George Trimponias, Ilaria Bartolini, Member, IEEE, Dimitris Papadias, and Yin Yang, "Skyline Processing on Distributed Vertical Decompositions". *IEEE Transactions On Knowledge And Data Engineering*, Vol. 25, No. 4, April 2013.

