Skyline with Presorting

Jan Chomicki[1] Parke Godfrey[2,3] Jarek Gryz[3] Dongming Liang[3]

[1]University at Buffalo
Buffalo, NY 14260-2000 USA
chomicki@cse.buffalo.edu

[2]The College of William and Mary
Williamsburg, VA 23187-8795 USA
godfrey@cs.wm.edu

[3]York University
Toronto, ON M3J 1P3 CANADA
{jarek, liang}@cs.yorku.ca

Abstract

The skyline, or Pareto, operator selects those tuples that are not dominated by any others. Extending relational systems with the skyline operator would offer a basis for handling preference queries. Good algorithms are needed for skyline, however, to make this efficient in a relational setting. We propose a skyline algorithm, SFS, based on presorting that is general, for use with any skyline query, efficient, and well behaved in a relational setting.

1 Introduction

In [2], the skyline operator, and a corresponding skyline of clause, were proposed, for the relational engine and for SQL, respectively.

```
select ... from ... where ...
      group by ... having ...
      skyline of a1 [min | max | diff], ...,
                 an [min | max | diff]
```

Skyline chooses each tuple which is not *dominated by* any other tuple; that is, no other tuple is better over each of the designated attributes. The diff directive works as a group by within the skyline, and the skyline for each group of diff attributes' values is found. The notion of skyline is the same as that of finding the *Pareto optimal* tuples with respect to criteria. Thus, this is also called the Pareto operator.

Skyline has prompted much interest recently, as people see its potential as a basis for—or at least a major component of—efforts to provide preference queries over relational database systems [1, 3, 6, 7, 8]. Skyline queries can be translated into current SQL, but the resulting queries are inefficient to execute. To handle skyline queries well, the skyline operator would have to be built into the query optimizer, and good algorithms for skyline implemented.

We are seeking to devise algorithms for skyline with good performance, which are well-behaved in a relational setting, and which will work in all general cases. In [9, 10], skyline algorithms that exploit indexes are introduced. They are not general though, as the index structures are particular to skyline, they will only help for specific queries but not others, and they are not composable with other relational operations as selections. In [2], two basic algorithms are explored: a divide-and-conquer-based approach (D&C); and a block-nested loops approach (BNL). D&C was shown to be preferable for higher dimensional (over five) queries. However, BNL was better for the mid-range of dimensions, and better uniformally on I/O. D&C would not scale well for larger datasets—or smaller buffer pools—than used in [2]. This leaves BNL as the only viable option of the approaches explored as a general, relational algorithm for skyline. We show that BNL can be revised via presorting to build a more effective algorithm. We call our algorithm *sort-filter-skyline*, SFS. Please see [4] for a full version of this paper. In [5], we explore cardinality estimation for skyline queries, which would be needed in the query optimizer's cost model.

2 Sort-Filter-Skyline Algorithm

In [2], a multi-pass algorithm for skyline computation is introduced, *skyline BNL*, which is reminiscent of the basic *block-nested-loops join* algorithm. BNL works as follows. We start with the table of tuples from which the skyline is to be determined. This may, of course, be an intermediary, temporary table $\mathbf{T}_0$ created during the SQL query employing the skyline clause.

BNL keeps a *window* in the buffer pool for collecting candidate skyline tuples. It commences with $\mathbf{T}_0$ as the input of the initial *pass*. The window is initially empty. (Let us refer to the current pass by the input that is being currently processed, say $\mathbf{T}_i$.) A page of tuples is read from $\mathbf{T}_i$. Each tuple, Q, from the page is compared with each tuple, S, in the window. If S dominates Q, then Q is discarded. If Q dominates S, then S is discarded from the window. Q is compared with the rest of the window tuples to see what else Q dominates and hence is discarded. Q is then added to the window itself as a candidate skyline tuple. Otherwise, Q is incomparable with all the tuples in the window. In this case again, Q should be added to the window itself as a candidate skyline tuple. If there is room in the window, Q is added. Otherwise, Q is written to temporary table $\mathbf{T}_{i+1}$.

Once BNL comes to the end of $\mathbf{T}_i$, some of the skyline tuples can be identified. If $\mathbf{T}_{i+1}$ is empty, the algorithm is finished. All tuples in the window are skyline. Otherwise, those tuples which were written to the window before the first tuple was written to $\mathbf{T}_{i+1}$ are skyline. BNL removes and reports those, but leaves the remaining tuples in the window for the next pass. The process is started again with $\mathbf{T}_{i+1}$ as the current input.

The skyline BNL in [2] makes an optimization to the description above: once a tuple in the window has gone through a full "cycle" of comparisons, it is output as a skyline tuple. There is no need to wait to the end of the pass for each S. This can be done with appropriate bookkeeping.

The I/O expense of BNL depends on the number of passes made *and* the size of the passes. The size of first pass, $\mathbf{T}_0$, is clearly the full initial input table. Unlike the join BNL, the skyline BNL has an advantage in that each subsequent pass, $\mathbf{T}_{i+1}$, can become significantly smaller, since many tuples will be discarded during the current pass, $\mathbf{T}_i$. The performance of skyline BNL depends much on how efficient the discarding is. Note however, BNL has no means to affect the discarding effectiveness. If the allocated window is large enough to hold all the skyline tuples, BNL might work in a single pass. With larger (but still quite small) windows, though, BNL becomes CPU-bound rather than I/O-bound, because the window comparison operations are expensive.

Furthermore, BNL is *not* guaranteed to work in a single pass when the window is large enough to hold all the skyline tuples. For example, consider the input of $\langle 4,3\rangle$, $\langle 3,4\rangle$, $\langle 1,6\rangle$, $\langle 2,7\rangle$, and $\langle 5,5\rangle$, in that order. The number of skyline tuples, #S, is two ($\langle 2,7\rangle$ and $\langle 5,5\rangle$). Let the window size in number of tuples, #W, be two also. BNL needs two passes to find both. More generally, BNL is not guaranteed to complete in the optimal number of passes, $\lceil \#\mathsf{S}/\#\mathsf{W} \rceil$, and there are cases when BNL will take significantly more passes than optimal.

We sought to devise a more efficient skyline algorithm. For this, we exploited the following observation.

Theorem *Any total order of the tuples of* $\mathbf{R}$ *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.*

Our *sort-filter-skyline* algorithm, SFS, 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 $\mathbf{T}_0$. The algorithm proceeds as BNL, except now, when a tuple is added to the window during pass $\mathbf{T}_i$, we know that it is skyline. No tuple following it in $\mathbf{T}_i$ can dominate it, by the theorem above. Thus the tuple can be output as skyline immediately, and a copy placed in the window. Window operations in SFS are 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.
    - SFS is 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.

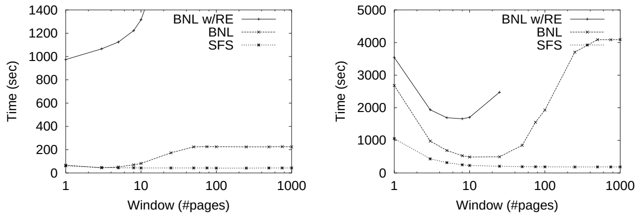
Since when a tuple is added to the window in SFS, it is known to be skyline, only the attributes involved in the skyline criteria need be projected. BNL must keep the entire tuple since it will not know until later whether it qualifies as skyline. Thus, SFS can keep many more (projected) tuples (#W) in the same-sized window than BNL. Since SFS has an initial sorting phase, we can control the effectiveness of discarding. In particular, we found that sorting by

$$E(t) = \sum_{i=1}^{k} \ln(t[\mathsf{a}_i] + 1)$$

where the a_i's are the normalized skyline conditions, yields the most effective discarding. This is sorting the tuples by an *entropy* function. The higher a tuple's entropy, the more tuples it likely dominates. Hence high-scoring skyline tuples will discard more tuples per pass. With sorting on E, SFS discards more effectively than BNL. More importantly perhaps is that we cannot control how an input table is ordered in a relational setting. (It is quite likely that the input is ordered somehow. All base tables will have a clustered index.) Since SFS necessarily presorts, the ordering of the input is immaterial. BNL does not presort, however, so is vulnerable to how the input is ordered. In fact, we show when the input is ordered from lowest to highest with respect to E (which we call RE for reverse entropy ordering), BNL has pathological performance. The RE ordering forces BNL to make exceedingly more passes than the optimal. For SFS, since any tuple that is added to the window is guaranteed to be skyline, SFS is guaranteed to perform in the optimal number of passes. Lastly, the use of diff in skyline of is beneficial for SFS. We shall presort the data based first on the diff attributes, then the other skyline attributes. SFS will flush the window whenever the diff attributes' values change. BNL cannot exploit this because the input is not sorted for it.

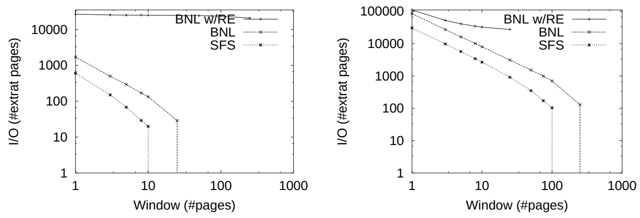
In addition to BNL's pathological cases, larger window allocations lead to CPU-boundedness and greatly diminished performance. SFS does not suffer from this; increased window allocation leads to increased performance to a point, and then levels off nicely.

We ran experiments over a million tuple table. Each tuple is 100 bytes, consisting of ten integer attributes (four



(a) 5 dim. (b) 7 dim.

Figure 1. Times for SFS versus BNL.



(a) 5 dim. (b) 7 dim.

Figure 2. I/O's for SFS versus BNL.

bytes each) and a sixty byte string, making for a 100MB dataset. A page for us is 4096 bytes, so 40 tuples fit per page. We use the integer columns for skyline dimensions. The data was randomly generated, the values are uniformally distributed, and the columns are pair-wise independent. Our testing program is written in C++. We implement both SFS and BNL in the same code-base for comparison. If-then-else statements switch to specifics for SFS or BNL from the main routine, as needed. We implement the basic BNL algorithm from [2]. We do not implement any of the optimizations discussed in [2], as the basic BNL and the optimized versions track quite closely in their experiments. We ran the experiments on a AMD Athlon 900-MHz PC with 384-MB main memory and a 40-gigabyte disk (7200-rpm, UDMA 100), running Microsoft Windows 2000.

Figure 1 shows timing results for BNL and SFS for the million tuple random dataset for a five- and a seven-dimension skyline query. The SFS algorithm used employs the optimizations discussed above: the table is pre-sorted with E; and the tuples for the window are projected to just the 40-bytes of the integers values. The sorting time for SFS *is included*. The external sort cost 37 seconds each time. BNL was run on the input in the order of the randomly generated data (BNL) and on the same input but sorted with RE (BNL w/ RE). Actual performance of BNL in a real relational environment could be expected to fall somewhere in-between. In Figure 1 (b), the line stops for BNL w/ RE as the times became too large to run to completion.

Figure 2 shows the same experiments, but reports the number of I/O's instead. The I/O's for the initial pass of the entire table and the I/O's to report the skyline are not counted, since these are shared by all algorithms. Where the lines become vertical is when the window is large enough to hold all the skyline.

3 Conclusions

SFS is a realistic algorithm for implementation of skyline in relational engines. There are numerous improvements that can be made to SFS, and we are pursuing better skyline algorithms based upon SFS. While there are advantages in that the sort of the data and the *filter-skyline* operation can be scheduled by the query optimizer separately, the sorting phase and the *filter-skyline* phase could be combined, as is done in the the standard *sort-merge join* algorithm. This would reduce overall the number of passes, increasing performance. It would be possible also to combine the D&C algorithm into the external sort passes to eliminate non-skyline tuples earlier on. This would lead to much smaller passes during the *filter-skyline* phase.

References

[1] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *SIGMOD*, pages 297–306, 2000.

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

[3] J. Chomicki. Querying with intrinsic preferences. In *EDBT*, 2002.

[4] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. Technical Report ??, Computer Science, York University, Toronto, ON, Canada, Oct. 2002. http://www.cs.yorku.ca/techreports/2002/CS-2002-??.html

[5] P. Godfrey. Cardinality estimation of skyline queries. Technical Report ??, Computer Science, York University, Toronto, ON, Canada, Oct. 2002. http://www.cs.yorku.ca/techreports/2002/CS-2002-??.html

[6] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, pages 259–270, 2001.

[7] W. Kießling. Foundations of preferences in database systems. In *VLDB*, Aug. 2002.

[8] W. Kießling and G. Köstler. Preference SQL: Design, implementation, experiences. In *VLDB*, Aug. 2002.

[9] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, Aug. 2002.

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