Evaluation of Skyline Algorithms in PostgreSQL

Hannes Eder
Institut für Informationssysteme
Technische Universität Wien
A-1040 Vienna, Austria
jeder@dbai.tuwien.ac.at

Fang Wei
Institut für Informatik
Albert-Ludwigs-Universität Freiburg
D-79110 Freiburg i. Br., Germany
fwei@informatik.uni-freiburg.de

ABSTRACT

In this paper, we present our work on evaluating the skyline algorithms BNL, SFS, and a variant of LESS in PostgreSQL. It is well known that the performance of skyline queries is sensitive to a number of parameters. From extensive experiments on skyline implementations we have discovered several rules, which are remarkably simple and useful, but hard to obtain from theoretical investigation. Our findings are beneficial for developing heuristics for the skyline query optimization, and in the meantime, provide some insight for a deeper understanding of the skyline query characteristics.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: DATABASE MANAGEMENT Systems

General Terms

Performance

Keywords

PostgreSQL, Skyline

1. INTRODUCTION

The skyline operator is important for supporting multi-criteria decision making applications. Given a set of d-dimensional data points, the skyline consists of the points, called skyline points, which are not *dominated* by another data point. Börzsönyi et al. [1] proposed a *skyline* extension of the SQL syntax with the form:

```
SELECT ... FROM ... WHERE ...
GROUP BY ... HAVING ...
SKYLINE OF [DISTINCT] a_1 [MIN|MAX|DIFF], ..., a_m
[MIN|MAX|DIFF]
ORDER BY ...
```

Since the introduction of the skyline operator [1], a number of secondary-memory algorithms have been developed for efficient skyline computation. These algorithms can be classified into two categories. The first one involves solutions that do not require any preprocessing on the underlying dataset. Algorithms such as *Block Nested Loops* (BNL) [1], *Divide and Conquer* (D&C) [1], *Sort First Skyline* (SFS) [3], and *Linear Elimination Sort for Skyline* (LESS) [5] belong to this category. The algorithms in the second category utilize different index structures such as sorted lists and R-trees to reduce the query costs. Well-known algorithms in this category include *Bitmap* [10], *Index* [10], *Nearest Neighbor* (NN) [7], and *Branch and Bound* (BBS) [8, 9].

In this paper, we present our work on evaluating skyline algorithms in PostgreSQL. Building the skyline query operation in an RDBMS has several advantages. First, skyline can be integrated with other relational operators, thus more sophisticated queries can be constructed. Moreover, no work on concurrency control and transaction management has to be addressed. Second, the existing index structures such as R-trees in an RDBMS can be adapted to implement index-based skyline algorithms. Third, if several skyline algorithms are available in an RDBMS, the system is able to select the most efficient one for the given datasets. Indeed, the comparisons of the skyline algorithms we have so far implemented reveal that there does not exist a clear winner in all aspects. The efficiency of the algorithms depend on the properties of the datasets such as dimension, distribution, cardinality and so on. For instance, BNL is more efficient than SFS and even LESS, if the dimension of the data is less than 5 and the number of tuples is relatively small.

We have so far implemented the non-index based algorithms BNL, SFS, and a variant of LESS in PostgreSQL. In addition, we extended the standard syntax to specify

- The treatment of NULL values (`NULLS FIRST` and `NULLS LAST`)
- The usage of order relations other than $<$ and $>$ (`USING` *`Op`*)
- Operational aspects of skyline computation, such as
  - *method* (BNL, SFS)
  - *tuple window size* in terms of memory and/or number of slots
  - *tuple window policy* (append, prepend, entropy, random)
  - *usage of indexes* (`NOINDEX`)
  - *usage of elimination filter (EF)*

The ultimate goal of our work is building a skyline query optimizer to automatically generate a good query plan w.r.t.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IDEAS 2009, September 16-18, Cetraro, Calabria [Italy]
Editor: Bipin C. DESAI
Copyright 2009 ACM 978-1-60558-402-7/09/09 ...$5.00.

I/O, time, and memory consumption. It is well known that the cost estimation of the skyline queries is a non-trivial task [2] since the performance of a skyline query is sensitive to a number of parameters [6]. To achieve this goal, we conducted extensive experiments on the skyline implementations.

We discovered several *hidden rules*, which are remarkably simple and useful, but hard to obtain from the theoretical investigation. We expect that our exposition of experimental results on skyline algorithms could serve as a guideline for developing heuristics of a skyline query optimizer, and in the meantime, provide some insight for a deeper understanding of the skyline query characteristics.

For the purpose of repeatability we have set up a website to present our experimental results. The source code, and the log-files from our experiments are accessible via `http://skyline.dbai.tuwien.ac.at/`.

2. SKYLINE ALGORITHMS

The skyline algorithms we have implemented are: BNL, SFS, BNL+EF and SFS+EF. EF stands for *Elimination Filter*, which was introduced in [5] as an essential routine for the LESS algorithm.

BNL was implemented according to the algorithm proposed in [1]. A *tuple window* is maintained in the main memory for storing the potential skyline tuples. Once the window becomes full, an overflow file has to be generated. In the PostgreSQL execution engine, the BNL operator lies in the iterator tree between the `ORDER BY` and the `GROUP BY` node (see Figure 1), to directly reflect the intended semantic of the `SKYLINE OF`-extension to the SQL syntax.

The implementation of SFS is according to [3]. It differs from BNL in that the data is topologically sorted at start-up time. We realize the sorting by utilizing the same physical operator as for the `ORDER BY`-clause. In the presence of a suitable index, an explicit sort phase can be omitted. While the skyline operator is insensitive to the order of attributes, more indexes can be considered by the query optimizer for the access path selection. We modified the query optimizer accordingly.

In [5] Godfrey et al. proposed the LESS algorithm, which is an improvement of SFS due to the introduction of the concept of *elimination filter* (EF). With the original LESS algorithm the elimination filtering is carried out in the pass zero of the external sort routine to eliminate records quickly.

In our case, in order to integrate the elimination filter into the SFS algorithm, while at the same time preserving the utilization of the physical sort operator, we implemented the elimination filter *before* the sort routine. Speaking in terms of the iterator tree, the EF node is a child of the sort node, which in turn is a child of the SFS node (see Figure 2(b)). Furthermore, we do not integrate the skyline computation into the final pass of the merge sort phase, as described in [5]. Therefore, technically speaking, SFS+EF is not an equivalent implementation of LESS. However, the elimination filter is preserved in SFS+EF, which is essentially the gist of LESS.

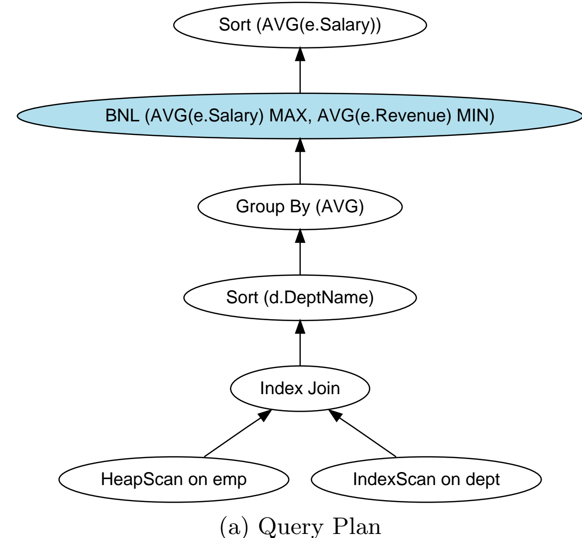
The implementation of EF is similar to BNL. An elimination filter window is maintained in the main memory. We use 8 KB as default window size. The difference between EF and BNL is that EF does not write tuples to a temporary file if the tuple window is full, and that the relative order of tuples going through the EF is preserved.

Inspired from the idea of the elimination filter in LESS, we experimented a new combination of BNL and EF. That is, the elimination filter is executed before the BNL algorithm. Since our implementation of the elimination filter is independent from the external sort routine, the coding of this variant is straightforward. It turns out that the algorithm BNL+EF is a substantial improvement to BNL.

We experimented with four window policies for both the tuple window (applied for BNL and SFS) and the EF window (for elimination filter): *append*, *prepend*, *entropy* and *random* [6]. We implemented both windows with doubly linked lists with a sentinel, thus the time cost for insertion and deletion was in $O(1)$. Furthermore, as the insert position for the *entropy* and *random* window policy was determined while iterating over the tuple window, the insert operation was also in $O(1)$.

If the EF window is full, in case of EF window policy *entropy* or *random*, the tuple with the lowest ranking is evicted to make space for a higher ranked tuple. Instead of calculating the entropy, a random value is used for ranking by the window policy *random*. With EF window policy *append* and *prepend*, once the EF window is full, space in the EF window is only gained if a tuple in the EF window is dominated by the incoming tuple and thus removed. In our implementation the window policy *entropy* can only be used if statistics for basetables is available.

In summary, we conducted experiments of the four skyline algorithms: BNL, SFS, BNL+EF, SFS+EF, upon each of which four window policies were applied: *append*, *prepend*, *entropy* and *random*. For BNL+EF and SFS+EF, we applied on both windows the same window policy in our experiments.



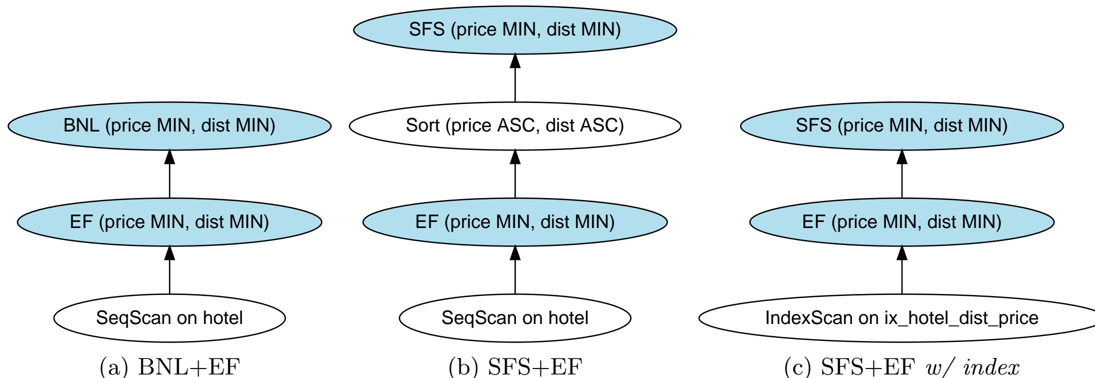
(a) Query Plan

```
SELECT d.DeptName, AVG(e.Salary), AVG(e.Revenue)
FROM emp e JOIN dept d on (e.Dno = d.DeptId)
GROUP BY d.DeptName
SKYLINE OF AVG(e.Salary) MAX, AVG(e.Revenue) MIN
ORDER BY AVG(e.Salary) DESC
```

(b) Query

Figure 1: Example query plan for BNL

3. EXPERIMENTAL SETUP



(a) BNL+EF (b) SFS+EF (c) SFS+EF *w/ index*

Figure 2: Query plans for: `SELECT * FROM hotel SKYLINE OF price MIN, dist MIN;`

We ran our experiments on seven Dell OptiPlex 755, Intel Pentium Dual-Core CPU E2160 (1.80GHz, 1MB L2 cache), with 1 GB RAM and Seagate Barracuda 160 GB HDD (7200 rpm, SATA 3.0Gb/s, 8 MB cache) with a single NTFS partition, running Microsoft Windows XP (SP2). Our implementation was based on PostgreSQL 8.3.0 and was compiled using Microsoft Visual Studio 2005 (SP1) using the `RELEASE` configuration with assertions disabled. PostgreSQL was configured to use 200 MB of RAM for shared buffers, with auto vacuuming disabled, and all other settings were left as default.

If not otherwise mentioned, a tuple window size of 1024 KB (equal to the PostgreSQL `work_mem` setting), and an EF tuple window size of 8 KB (equal to the PostgreSQL block size) were used.

The tables for the test runs have been generated using the extended version [4] of the dataset generated presented in [1], which allows to generate tables with different initial seeds for the random generator. All experiments were carried out on three different sets with varying initial seed, for each distribution type (independent, correlated, and anti-correlated), and with 100, 500, 1k, 5k, 10k, 50k, 100k tuples.

Each tuple consists of a unique 4 byte integer *id* and 15 randomly generated 8 byte floats $d_1, \ldots, d_{15}$. With a 23 byte tuple header and the alignment it sums up to a tuple length of 152 bytes. The memory chunk used by the PostgreSQL memory allocation function (`palloc`) was 264 bytes long.

For most of our experiments we did not consider tables with more than 100k tuples, as the runtime for a 15 dimensional skyline query on 100k tuples went up to 30 minutes.

All experiments were carried out with a *hot disk cache*, i.e. due to appropriate queries all pages were in the shared buffers.

4. COMPARISONS AND ANALYSIS

We measure the time performance with respect to three parameters: dimension, data cardinality, and data distribution. In this section we only report the experimental results on the data sets with *independent* distribution type. Other results on correlated and anti-correlated can be found in the full version of the paper. For the other two factors "dimension" and "cardinality", one has been fixed during each experiment. Thus all the plots are two dimensional. We deploy the relative values by setting one measurement as the reference.

Figure 3(a) shows the time performance of the skyline algorithms vs. dimension with 100k tuples. It shows clearly that the SFS+EF algorithm has the best time performance with all the data distributions. This conforms with the results in [6], where it was shown that LESS outperforms the SFS algorithm with 1M tuples.

BNL+EF performs consistently better than BNL, due to the low skyline selectivity factor for datasets with 100k tuples (cf. Figure 3(c)). As far as the window policy is concerned, *entropy* has consistently better performance for all the algorithms.

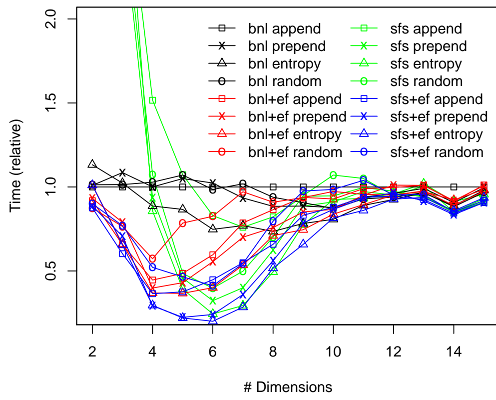
Figure 3(b) depicts the time-dimensionality performance regarding to datasets with 1k number of tuples. One interesting observation is that the performance of SFS is surprisingly better than others, except for the extremely low dimension values. As illustrated by Figure 3(c), there exists a causal relation with the skyline selectivity factors. It shows clearly that as the data cardinality decreases, the skyline selectivity factor increases. Therefore, in general, the EF operator is less effective with lower data cardinalities. To be more accurate, if the skyline selectivity factor is higher than 0.1, the benefit gained from EF is marginal. This is confirmed by the results shown in Figure 3(a) as well.

The result is somehow contradictory to the claim that SFS+EF should perform consistently better than SFS, because at the EF stage there should be always some tuples eliminated. However, one should not ignore the cost of the EF operation, where comparisons are executed. Moreover, if the *entropy* window policy is applied, the time consumption is even higher. Hence, there exist data settings where EF does not pay off anymore.

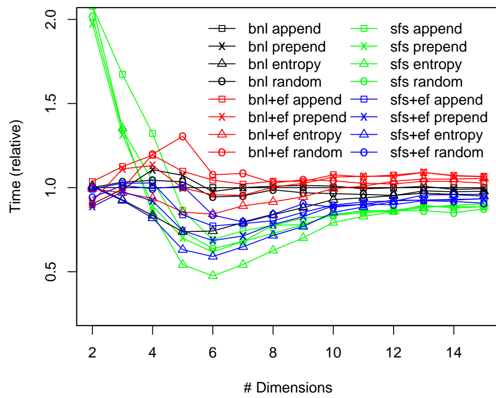
To better understand the time performance of the skyline algorithm with respect to the data cardinality, we conducted experiments for high dimensions of 15 with the number of tuples ranging from 100 to 100k. See Figure 3(d). Surprisingly, the *rule of the selectivity factor* is proven to be true in all the datasets. That is, if the selectivity factor is higher than 0.1, the elimination filter does not have any advantage.

As a matter of fact, if the dimension is higher than four, SFS is superior for all datasets, except for data cardinality of 50k and 100k, where SFS+EF outperforms SFS. If we examine again the selectivity factor for independent datasets in Figure 3(c), these datasets are in the category where the selectivity factor is higher than 0.1. This result confirms our observation. Note that the same observation holds for BNL+EF vs. BNL as well.

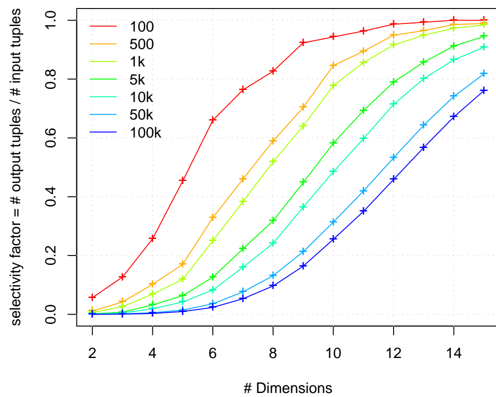
5. CONCLUSIONS AND FUTURE WORK



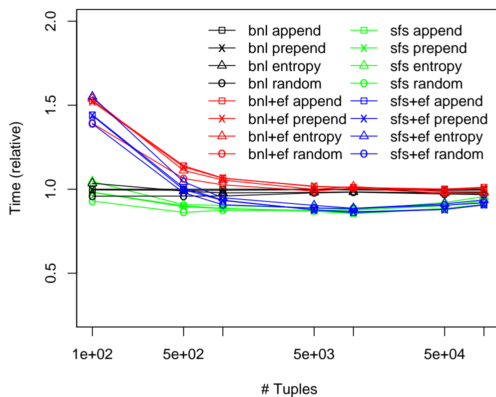
(a) Independent dataset (100k tuples)



(b) independent dataset (1k tuples)



(c) independent dataset



(d) 15 dim (independent)

Figure 3: Runtime for different methods

In this paper we presented the experiments with skyline algorithms applied on the open source RDBMS PostgreSQL.

From our experimental results we were able to expose several findings which are difficult to be verified theoretically. There are: (1) the elimination filter is effective only if the selectivity factor of the dataset is not more than 0.1; (2) for datasets up to 500 tuples and of relative small dimension (e.g. up to five), BNL performs the best in all aspects.

In the future work we plan to implement and evaluate index-based algorithms such as *Index* [10], *Nearest Neighbor* (NN) [7], and *Branch and Bound* (BBS) [8, 9]. We believe the concept of EF is very promising and plan to further improve its efficiency and effectiveness, e.g. to integrate it as a filter in the index scan code, to use other data structures and policies for the EF window, and to optimize the order of attributes when comparing two tuples.

As for SFS algorithms using indexes, what remains open is to investigate the behavior in case all skyline attributes and attributes in the select clause are part of the index, such that it suffices only to access the indexes for the query processing. We believe this could yield good results.

6. REFERENCES

[1] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *17th International Conference on Data Engineering (ICDE' 01)*, pages 421–432, Heidelberg, Germany, 2001. IEEE.

[2] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 64, Washington, DC, USA, 2006. IEEE Computer Society.

[3] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 717–816. IEEE Computer Society, 2003.

[4] H. Eder. Random dataset generator for skyline operator evaluation. `http://randdataset.projects.postgresql.org/`, 2007.

[5] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, and B. C. Ooi, editors, *VLDB*, pages 229–240. ACM, 2005.

[6] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB J.*, 16(1):5–28, 2007.

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

[8] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *SIGMOD Conference*, pages 467–478. ACM, 2003.

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

[10] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 301–310. Morgan Kaufmann, 2001.