Group-by Skyline Query Processing in Relational Engines

Ming Hay Luk   Man Lung Yiu   Eric Lo

Hong Kong Polytechnic University
{csmhluk, csmlyiu, ericlo}@comp.polyu.edu.hk

ABSTRACT

The skyline operator was first proposed in 2001 for retrieving interesting tuples from a dataset. Since then, 100+ skyline-related papers have been published; however, we discovered that one of the most intuitive and practical type of skyline queries, namely, *group-by skyline queries* remains unaddressed. Group-by skyline queries find the skyline for each group of tuples. In this paper, we present a comprehensive study on processing group-by skyline queries in the context of relational engines. Specifically, we examine the composition of a query plan for a group-by skyline query and develop the missing cost model for the BBS algorithm. Experimental results show that our techniques are able to devise the best query plans for a variety of group-by skyline queries. Our focus is on algorithms that can be directly implemented in today's commercial database systems without the addition of new access methods (which would require addressing the associated challenges of maintenance with updates, concurrency control, etc.).

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing, relational databases*

General Terms

Design, Performance

1. INTRODUCTION

The skyline operator, which returns a set of tuples not dominated by any other tuple, is important for many multi-criteria decision making applications. Since its introduction in [1], numerous variants of the operator have been proposed. Surprisingly, if we review the original skyline specification [1] (see below), there is one very important and basic part of the specification, hitherto unaddressed:

```
SELECT ...      FROM ...        WHERE ...
GROUP BY ...    HAVING ...
SKYLINE OF      d_1 [MIN|MAX], ..., d_m [MIN|MAX]
ORDER BY ...
```

Specifically, the processing of skyline queries with the GROUP BY clause, has not yet been well addressed in the database community. This paper is devoted to studying this problem in relational engines.

The *group-by skyline query problem* can be formally defined as follows. Let $\mathcal{D}$ be a relational table instance, with schema $A = (A_1, A_2, \ldots, A_k)$. The notation $t[A_i]$ represents the value of a tuple t in the attribute A_i. Given a set $\mathcal{G} \subset A$ of grouping attributes and a group instance g of $\mathcal{G}$, we define the set $\mathcal{D}(g)$ as the set of tuples of $\mathcal{D}$ belonging to the group g:

$$\mathcal{D}(g) = \{t \in \mathcal{D} \mid \forall\ A_i \in \mathcal{G},\ t[A_i] = g[A_i]\}$$

Given a set $\mathcal{S} \subset A$ of skyline attributes, a tuple t is said to *dominate* another tuple t', denoted by $t \succ_{\mathcal{S}} t'$, if:

$$(\exists\ A_i \in \mathcal{S},\ t[A_i] < t'[A_i]) \wedge (\forall\ A_i \in \mathcal{S},\ t[A_i] \leq t'[A_i])$$

assuming that smaller values are preferable over larger ones. The result set of a skyline query is defined as:

$$\Psi(\mathcal{D}, \mathcal{S}) = \{t \in \mathcal{D} \mid \nexists t' \in \mathcal{D}, t' \succ_{\mathcal{S}} t\}$$

In other words, a tuple t belongs to the skyline result set if no other tuple dominates it.

Given the sets $\mathcal{G}$ and $\mathcal{S}$ of attributes such that $\mathcal{G} \cap \mathcal{S} = \emptyset$, the *group-by skyline* query $Q = (\mathcal{G}, \mathcal{S})$ computes a skyline result set $\Psi(\mathcal{D}(g), \mathcal{S})$ for each group instance g defined on $\mathcal{G}$. We use $Q(\mathcal{D})$ to denote the overall result set of Q on the dataset $\mathcal{D}$.

Table 1a shows an exemplary dataset $\mathcal{D}$ of the scores of students in a school. Each tuple represents a student, with its 'Grade' and 'Gender', as well as the scores of subjects in 'English', 'Math', and 'Science'. In the group-by skyline query:

```
SELECT Grade, Student, English, Math
FROM    D
GROUP BY Grade
SKYLINE OF English MAX, Math MAX
```

a user (e.g., a teacher) specifies the grouping attributes as the set $\mathcal{G} = \{\text{Grade}\}$, and the skyline attributes as the set $\mathcal{S} = \{\text{English, Math}\}$. The keyword MAX indicates that higher values are preferred to lower ones in those skyline attributes. Conceptually, the query partitions the table $\mathcal{D}$ into groups according to $\mathcal{G}$, and then computes the skyline tuples of each group with respect to $\mathcal{S}$.

Observe that a traditional skyline query without the GROUP BY clause on the above dataset may not be meaningful because it is unfair to compare a student from grade 07 with a student from grade 08. On the other hand, the group-by skyline query computes the outstanding students for each grade, providing a much more intuitive and meaningful result.

The goal of this paper is to present the nuts and bolts for supporting group-by skyline queries in *relational query engines*. Given that there is a wealth of solutions in skyline query processing, we

CIKM'09, November 2–6, 2009, Hong Kong, China.
Copyright 2009 ACM 978-1-60558-512-3/09/11 ...$10.00.

Table 1: Example

Student	Grade	Gender	English	Math	Science
a	7	M	50	50	50
b	7	M	100	25	100
c	7	M	70	70	70
d	7	F	45	40	60
e	7	F	75	60	80
f	8	F	60	50	30
g	8	F	40	30	20

(a) School Database

Grade	Student	English	Math
07	b	100	25
	c	70	70
	e	75	60
08	f	60	50

(b) Group-by Skyline Result

emphasize that we are not going to duplicate those efforts. Instead, we try to utilize the existing techniques whenever they are applicable. However, if any issues remain open, we will provide technical solutions in their respective sections.

In the relational engine setting, the query optimizer needs to perform *cost estimation* for different query plans and then select the best one. A recent paper [2] follows this traditional cost-based principle and provides cost estimation equations (i.e., skyline cardinality, I/O cost, and CPU cost) for two skyline algorithms, Block-Nested-Loop (BNL) [1] and Sort-Filter-Skyline (SFS) [3]. We follow this line and devise the cost model for the Branch and Bound Skyline (BBS) algorithm [9], which turns out to be a useful component for processing our group-by skyline query as well. In fact, we are also aware of recent work on efficient skyline algorithms (e.g., [6, 14]); however, in this paper, we first focus on algorithms that can be directly implemented in today's commercial database systems without the addition of new access methods (which would require addressing the associated challenges of maintenance with updates, concurrency control, etc.).

2. COMPOSING A QUERY PLAN WITH OPERATORS

The *R-tree Group-by Skyline Algorithm* (RGS) [9] is the only group-by skyline algorithm to date. It operates on an R-tree which is built on all the attributes (grouping and skyline attributes) of a data set $\mathcal{D}$. Unfortunately, the RGS algorithm has several deficiencies in processing group-by skyline queries such as: (i) huge main memory consumption, (ii) inapplicability for ad-hoc queries, and (iii) performance degradation due to additional dimensions in the R-tree.

The first advantage of forming a query plan with operators over using RGS is that after estimating the cost of each feasible query plan, the query optimizer has the flexibility to choose the plan with the lowest cost to execute [7]. Furthermore, memory thrashing seldom occurs because only one group of tuples is retrieved from the preceding grouping operator and it remains within main memory for skyline processing.

To enable the query optimizer to decide which is the best group-by skyline query plan, we provide a comprehensive cost analysis of each implementation.

The traditional query evaluation method combines a *grouping operator* followed by a *skyline operator* to form a so-called *group-by skyline evaluation plan*.

For the grouping operator, we consider using recursive hashing and sorting [4] when no index is available. Additionally, since no index is available, we consider using the BNL algorithm [1] and the SFS algorithm [3] for the skyline operator. In the case of indexed data, we only consider the BBS algorithm [9] as it is I/O-optimal.

2.1 The Grouping Operation

We first review and reformulate the cost model for the grouping operator. As the grouping operator is an I/O-bound operation [4, 11], we focus on its I/O cost (as disk pages).

2.1.1 No Index: Sorting

When the grouping operation is implemented by sorting, the I/O cost (disk page accesses) is:

$$PA_{SORT}(N,B,M) = \frac{N}{B} \cdot \left(1 + 2 \cdot \lceil \log_{M-1} \frac{N}{B \cdot M} \rceil\right) \quad (1)$$

where N denotes the number of data points, M the number of memory pages, and B the number of tuples per disk page [4, 11].

2.1.2 No Index: Hashing

The I/O cost of recursive hashing depends on the number of distinct groups in a dataset. Let $|\mathcal{G}|$ be the number of grouping dimensions and β_i be the number of distinct values in the i-th grouping dimension. The number of all possible groups is thus $\prod_{i=1}^{|\mathcal{G}|} \beta_i$. Under uniform distribution assumption of groups, the number of distinct groups ω in the dataset is [10]:

$$\omega = \prod_{i=1}^{|\mathcal{G}|} \beta_i \left(1 - \left(1 - \frac{1}{\prod_{i=1}^{|\mathcal{G}|} \beta_i}\right)^N\right) \quad (2)$$

To simplify our cost equations, we assume that β_i is the same (say, β) for all grouping dimension. We also assume that the dataset is sufficiently large, i.e., $N \gg \beta^{|\mathcal{G}|}$, therefore we have:

$$\omega \approx \beta^{|\mathcal{G}|} \quad (3)$$

As a remark, if the data is non-uniform, the number of distinct groups ω can be estimated using the probabilistic counting technique of [10] using only a single pass of data. Substituting Equation 3 into the I/O cost equation of the recursive hashing algorithm given in [11], we can obtain the I/O cost of recursive hashing as:

$$PA_{HASH}(N,B,M) = 2 \cdot \frac{N}{B} \cdot \lceil \log_{M-1} \omega - 1 \rceil \quad (4)$$

2.1.3 With Index

In case a disk-based index (e.g., hash index, B$^+$-tree, R-tree) of the dataset on the attributes $\mathcal{G}$ is available, an online grouping algorithm can be applied to separate the indexed tuples into groups. This incurs an I/O cost of:

$$PA_{GROUP-INDEX} = \frac{N}{B} \quad (5)$$

2.2 The Skyline Computation

Skyline computation is a CPU intensive operation [1, 2], so we consider not only its I/O cost but also its CPU cost (as number of comparisons).

To process group-by skyline queries, we need to invoke a skyline algorithm over tuples within the same group, for each of the ω groups returned by the preceding grouping operator. In the following, we present the I/O cost and CPU cost of BBS [9]. The CPU costs of BNL and SFS are studied by [2] already and are reformulated for group-by skyline query processing in [7]. Additionally, the I/O cost for BNL and SFS are also detailed in [7].

2.2.1 With Index: BBS

The BBS algorithm [8] is an R-tree-based skyline algorithm. It performs in a similar way as the RGS algorithm but is I/O optimal. BBS maintains a heap when traversing the R-tree such that it always evaluates and expands the entry that is closest to the origin among all unvisited entries. Initially, it inserts the root of the R-tree into the heap. Entries in the heap are organized according to the $mindist$ function such that entries that are closer to the origin will be deheaped first. In each iteration, the top element e is deheaped and examined against the skyline computed so far. If e is not dominated by any current skyline objects, either e is output as a new skyline object (if e is an object) or the child entries of e are inserted into the heap (if e is an intermediate node). BBS terminates when the heap is empty.

Now we develop the cost model of BBS. For group-by skyline processing, it is necessary to build an R-tree on the attributes of $\mathcal{S}$ for each group of tuples produced by the preceding grouping operator. This can be implemented by an R-tree bulk-loading algorithm [5], which uses a space filling curve to sort the data points and then sequentially pack them into the tree. This includes the cost of one external sorting on the data set. As a result, for ω groups of tuples, the R-tree bulk-loading algorithm incurs an I/O cost R^{tree}_{load} of:

$$R^{tree}_{load} = \omega \cdot PA_{SORT}(\frac{N}{\omega}, B, M) \tag{6}$$

Next, we need to estimate the I/O cost and CPU cost of BBS. As each R-tree contains N/ω tuples, by [13], we derive the height of the tree equal to: $\lceil \log_f \frac{N}{\omega} \rceil$, where f denotes the average R-tree fanout. For the moment, we consider the nodes at the ith-level (e.g., the leaf nodes are at the 0-th level). By applying the model of [13], the number of nodes in the i-th level equals to $N/(f^{i+1} \cdot \omega)$. Let λ_i be the side length of a node in the i-th level. Since the dimensionality of the tree is $|\mathcal{S}|$, the volume of each node is $\lambda_i^{|\mathcal{S}|}$. Assuming that the nodes at the same level do not overlap, the total volume of all nodes equals 1. Thus, we express λ_i as:

$$\frac{N}{f^{i+1} \cdot \omega} \cdot \lambda_i^{|\mathcal{S}|} = 1 \Longrightarrow \lambda_i = \left(\frac{f^{i+1} \cdot \omega}{N}\right)^{\frac{1}{|\mathcal{S}|}} \tag{7}$$

Figure 1 illustrates the R-tree nodes at the i-th level. The side length of each node is λ_i. According to the search order of BBS, the white node will be visited first. The white node contains at least one skyline point, which is guaranteed to dominate any point in any dark-gray node. In other words, all dark-gray nodes will be pruned. However, the light-gray nodes cannot be pruned as they may contain some skyline point. As a result, the BBS algorithm accesses only the white node and light-gray nodes at the level i. Since the dimensionality of the tree is $|\mathcal{S}|$, the fraction of node accesses is equal to: $1 - (1 - \lambda_i)^{|\mathcal{S}|}$. By summing the above cost for all tree levels, for all ω trees, we derive the I/O cost of BBS as:

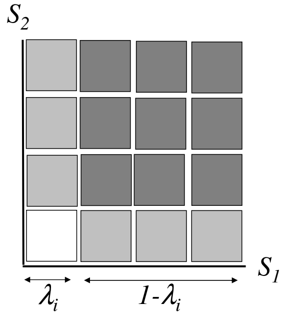

Figure 1: Node Accesses of BBS algorithm over an R-tree

$$\begin{aligned} & PA_{BBS}(N, f, \mathcal{S}, \omega) \qquad (8) \\ = \; & R^{tree}_{load} + \sum_{i=0}^{\lceil \log_f \frac{N}{\omega} \rceil - 1} \omega \cdot \left(\frac{N}{f^{i+1} \cdot \omega}\right) \left[1 - (1 - \lambda_i)^{|\mathcal{S}|}\right] \\ = \; & R^{tree}_{load} + \sum_{i=0}^{\lceil \log_f \frac{N}{\omega} \rceil - 1} \frac{N}{f^{i+1}} \cdot \left[1 - \left(1 - \left(\frac{f^{i+1} \cdot \omega}{N}\right)^{\frac{1}{|\mathcal{S}|}}\right)^{|\mathcal{S}|}\right] \end{aligned}$$

We continue to study the computation cost of BBS, in terms of the number of dominance comparisons. BBS performs dominance comparisons between the currently examined entry e and the set of skyline points seen so far. The entry e can either be a non-leaf entry or a data point. Since the total number of data points is far greater than that of non-leaf entries, we ignore the dominance comparisons contributed by non-leaf entries.

When the entry e is a data point, it suffices to find the number of dominance comparisons for tuples in the same group. The min-heap H in the BBS algorithm essentially rearranges the data points in a sorted order, so BBS behaves similarly to SFS. The only difference is that, in BBS, the tuple of each pruned leaf node will not be compared with the tuples of any pruned leaf node. According to our previous discussion, the fraction of pruned nodes equals to: $(1 - \lambda_0)^{|\mathcal{S}|}$. Therefore, the number CP_{BBS} of dominance operations in BBS can be approximated as:

$$\begin{aligned} & CP_{BBS}(N, \mathcal{S}, \omega) \qquad (9) \\ \approx \; & CP_{SFS}(\omega, N, |\mathcal{S}|) - CP_{SFS}(\omega, N \cdot \left(1 - \left(\frac{f\omega}{N}\right)^{\frac{1}{|\mathcal{S}|}}\right)^{|\mathcal{S}|}, |\mathcal{S}|) \end{aligned}$$

The number CP_{SFS} of dominance comparisons of SFS can be estimated by the proposal of [2], which requires skyline cardinality estimation [2][8].

3. EXPERIMENTS

In this section, we first present the settings of our experiments. Then, we present the results of our techniques for processing group-by skyline queries.

3.1 Experimental Settings

All the experiments where conducted on an Intel Core 2 Duo 2.4 GHz PC with 2GB of memory. As real database systems seldom have all its memory allocated to a single query, we set the amount of main memory available to a query as at most 10% of the size of the input. The page size is set to 4K Bytes. For all the R-tree based methods (e.g., BBS), the fanout is set to 81.

Table 2: Experimental Results

Varying N	Actual			Estimated		
	Ind	**Anti**	**Corr**	**Ind**	**Anti**	**Corr**
I/O winner	IDX-BNL	IDX-BNL	IDX-BNL	IDX-BNL	IDX-BNL	IDX-BNL
I/O loser	SRT-SFS	SRT-SFS	SRT-SFS	SRT-SFS	SRT-SFS	SRT-SFS
CPU winner	*BBS	*SFS	*BBS	*BBS	*BBS	*BBS
CPU loser	*BNL	*BNL	*BNL	*BNL	*BNL	*BNL

(a) Winners and losers of plans on synthetic data; varying N from 100K to 500K tuples; fixing $|\mathcal{G}|$=2, $|\mathcal{S}|$=3

Varying $|\mathcal{S}|$	Actual			Estimated		
	Ind	**Anti**	**Corr**	**Ind**	**Anti**	**Corr**
I/O winner	IDX-BNL	IDX-BNL	IDX-BNL	IDX-BNL	IDX-BNL	IDX-BNL
I/O loser	SRT-SFS	SRT-SFS	SRT-SFS	SRT-SFS	SRT-BBS	SRT-SFS
CPU winner	*BBS	*SFS	*BBS	*BBS	*SFS	*BBS
CPU loser	*BNL	*BNL	*BNL	*BNL	*BNL	*BNL

(b) Winners and losers of plans on synthetic data; varying $|\mathcal{S}|$ from 2 to 5; fixing $|\mathcal{G}|$=2, N=100K tuples

We have carried out experiments on both real data and synthetic data. The real data is the NBA players' technical statistics from 1946 to 2007. It contains 20,788 tuples in total, where each tuple stores the statistics of a player in a season, containing: two attributes for group-by, and eight attributes for skyline. : games played (gp), points (pt), rebounds (reb), assists (ast), steals (stl), blocks (blk), free throws (ftm), three-point shots (tpm). We considered queries on the following attributes (for synthetic datasets): *Ind* (independent attributes), *Corr* (correlated attributes), and *Anti* (anti-correlated attributes). Unless stated otherwise, each dataset contains $N = 100000$ tuples, with a total of 20 attributes (a_1, a_2, …, a_{20}): attributes $a_1 \ldots a_5$ are for grouping (with domain size $\beta = 5$), attributes $a_6 \ldots a_{20}$ are for skyline (with domain size $\theta = 10000$). Specifically, $a_6 \ldots a_{10}$ are generated independently, $a_{11} \ldots a_{15}$ are correlated to a_6, and $a_{16} \ldots a_{20}$ are anti-correlated to a_6. By default, each query has $|\mathcal{G}| = 2$ group-by attributes and $|\mathcal{S}| = 3$ skyline attributes.

3.2 Cost Model Evaluation

From the results in all our experiments, RGS was found to incur an extremely high number of I/Os because it demanded an R-tree to be built for all 20 attributes of the dataset and the prohibitive number of entries/data points forced the min-heap to be placed on disk rather in the memory. In the case of anti-correlated datasets, RGS could not terminate within hours. Due to RGS's impracticability, we omit RGS from the subsequent discussion.

In the following, a plan is said to be a winner (or loser) if it incurs the lowest (or highest) cost for the majority of tested cases. Similarly, we compute the estimated cost of each plan, and determine the winners/losers according to the estimation.

Effect of the Data Size N. As the CPU cost is independent of the group-by formation, we use *BBS to represent all plans involving BBS (asterisk being a wildcard). Table 2a shows that the estimation is able to predict the winners/losers correctly in 11 out of 12 cases.

Although BBS is I/O-optimal for computing skyline, it does not excel above BNL in the context of group-by skyline query processing because it requires building R-trees at query-time. However, BBS-related plans are effective in pruning because they are the CPU winners in most cases. For anti-correlated data, *BBS is not able to prune R-tree nodes effectively so its actual CPU cost is very close to *SFS.

Effect of the Number of Skyline Attributes $|\mathcal{S}|$. In Table 2b, we see that the estimated winners/losers are almost identical to the actual winners/losers. In the case of I/O cost in anti-correlated data, the actual costs of SRT-SFS and SRT-BBS are quite small at large $|\mathcal{S}|$. Thus, the incorrect prediction is not a problem in this case.

Effect of the Number of Group-by Attributes $|\mathcal{G}|$. Our technique is able to estimate winners/losers correctly in all tested cases.

Results on Real Data. We then investigate the effect of group-by and skyline attributes on the NBA dataset. Our experiments show that the estimated winners/losers on I/O cost and CPU cost all match with the actual ones. In fact, the correctness of prediction is robust in all cases. The above results confirm that it is practical to apply the estimation model even on real datasets.

4. CONCLUSIONS

This paper studies the processing of group-by skyline queries in relational databases. We show that processing group-by skyline queries by composing relational evaluation plans is much more efficient than using a single holistic algorithm. We further provide cost estimation techniques for implementing group-by skyline queries in relational engines.

Experimental results on both real and synthetic data suggest that we are able to, in most cases, predict the winner/loser query plans for group-by skyline queries.

As future work, we will upgrade our estimation models using the kernel-based skyline cardinality estimation techniques from [15]. There are other algorithms such as [6, 12, 14] that use specialized data structures. We leave the consideration of those algorithms as future work.

5. REFERENCES

[1] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, 2001.

[2] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust Cardinality and Cost Estimation for Skyline Operator. In *ICDE*, 2006.

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

[4] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[5] I. Kamel and C. Faloutsos. On Packing R-trees. In *CIKM*, 1993.

[6] K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the Skyline in Z Order. In *VLDB*, 2007.

[7] M.-H. Luk, M. L. Yiu, and E. Lo. Group-by Skyline Query Processing in Relational Engines. In *Hong Kong Polytechnic University. Technical Report*, 2009.

[8] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD*, pages 467–478, 2003.

[9] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *TODS*, 30(1):41–82, 2005.

[10] A. Shukla, P. Deshpande, J. F. Naughton, and K. Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. In *VLDB*, 1996.

[11] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, fifth edition, 2005.

[12] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *VLDB*, 2001.

[13] Y. Theodoridis and T. K. Sellis. A Model for the Prediction of R-tree Performance. In *PODS*, 1996.

[14] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable Skyline Computation Using Object-based Space Partitioning. In *SIGMOD*, 2009, to appear.

[15] Z. Zhang, Y. Yang, R. Cai, D. Papadias, and A. Tung. Kernel-Based Skyline Cardinality Estimation. In *SIGMOD*, 2009.