Skyline Evaluation Within Join Operation, Block Nested Loop Join Implementation

Ms. Sunita Dwivedi  
Makhanlal Chaturvedi National University of Journalism and Communication, Bhopal (M.P.) India

Dr. Anil Rajput  
Department of Mathematics and Computer Science  
Chandra Shekhar Azad Govt.P.G.Nodal College  
Sehore (M.P.), India

Abstract - Skyline Join approach in its Naïve age work as it computes join first and then apply skyline computation to find corresponding skyline objects. Considering increase in cardinality and dimensionality of join table the cost of computing skyline in a non-reductive join relation is costlier than that of on single table. Most of the existing work on skyline queries for databases mainly discusses the computation efficiency in single relational table. In proposed work we investigate the evaluation of skylines over disparate sources via joins in efficient manner. The basic idea of our approach is that without computing skyline in the entire joined table, we can process the joined skyline only based on the property of being skyline to quickly identify the skyline object for the joined tuple. We propose an algorithm to build on top of the traditional relational Block Nested Loop join algorithms, which fuses the computation of the join and the skyline in order to outputs the correct skyline without computing the full join. Our Experimental results demonstrate the applicability of interweaving join and skyline together.

Keywords : Skyline Join, Skyline Objects, Non Reductive, Block Nested Loop Join, Cardinality, Dimensionality

I INTRODUCTION

In the presence of the huge amounts of data that today’s systems are providing access to, it is a tedious task for a user to find the most interesting available data without using advanced query types, such as skyline queries. The skyline operator was first introduced in [3], where the authors extended SQL’s SELECT statement by an optional SKYLINE OF clause, such that the user can specify the dimensions as well as the function (MIN, MAX, DIFF) used for the skyline query. It is a common assumption to view skyline as an add-on operator on top of the traditional SPJ queries. Most of the existing work on skyline queries for databases mainly discusses the computation efficiency in single relational table, and treat skylines as an additional operator which is not part of query plan at early stages at completion of query execution under join, and group-by operators and before a final sort operator skyline evaluation is performed. Even though it is conceptually possible to apply these techniques on the results of a join operator, this solution is not likely to perform well in practice.

In general, an SQL-like statement for expressing a skyline join query based on the SKYLINE OF clause [3] is given as follows:

```
SELECT R_i.D_j | AGG (R_i.D_j) FROM R_1, R_2, ..., R_m
WHERE join condition (R_1, R_2, ..., R_m) GROUP BY... HAVING...
SKYLINE OF R_i.D_{j1} [DISTINCT] [MIN|MAX|DIFF], ..., R_m.D_{jnm} [MIN|MAX|DIFF]
ORDER BY...
```

When Join is performed cardinality and dimensionality of join relation increases potentially [6], [7] so the cost of finding skylines in the joined table will be even larger. Joining tables leads to increase number of attributes as join is assumed to be Non Reductive [4]. This can be expressed in example below.

Example 1.1
Consider two tables (I,II) CUSTOMER (CNum, Age, Account Balance) and part order table ORDER (ONum, CNum, PNum, Quantity, Amount)
Let following questions encounter : Find young customers with high account balance and having one order with high quantities of parts and high amount of price?

To handle the above we assume:
- Skyline operator as proposed [3] is part of SQL.
- Attributes CNum of C and CNum of O are join attributes.
- Attributes Age, Account Balance, Quantity and Amount are descriptive attributes participating in skyline evaluation.
- We assume only descriptive attributes, with numeric domain are participating in skyline.
The join results and the skyline objects in Customer $\bowtie$ Order are shown in Table III. Again we observe that the joined skylines may contain tuples that are not in the skyline of the individual input tables, such as the tuple with $D_1 = 105$.

### Table II

**ORDER table**

<table>
<thead>
<tr>
<th>ONum</th>
<th>CNum</th>
<th>PNum</th>
<th>Quantity</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101</td>
<td>001</td>
<td>1</td>
<td>274</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>001</td>
<td>6</td>
<td>1664</td>
</tr>
<tr>
<td>3</td>
<td>102</td>
<td>002</td>
<td>10</td>
<td>1999</td>
</tr>
<tr>
<td>4</td>
<td>103</td>
<td>003</td>
<td>1</td>
<td>400</td>
</tr>
<tr>
<td>5</td>
<td>104</td>
<td>004</td>
<td>5</td>
<td>900</td>
</tr>
<tr>
<td>6</td>
<td>104</td>
<td>004</td>
<td>6</td>
<td>1080</td>
</tr>
<tr>
<td>7</td>
<td>105</td>
<td>005</td>
<td>2</td>
<td>1900</td>
</tr>
</tbody>
</table>

The SQL statements for answering the above questions are as follows:

```sql
SELECT * FROM Customer C, Order O
WHERE C.CNum= O.CNum
SKYLINE OF G.Age MIN, C.AccountBalance MAX,
O.Quantity MAX, O.Amount MAX;
```

### Table III

**Joined Table of CUSTOMER and ORDER**

<table>
<thead>
<tr>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
<th>D7</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>35</td>
<td>90</td>
<td>1</td>
<td>101</td>
<td>1</td>
<td>274</td>
</tr>
<tr>
<td>101</td>
<td>35</td>
<td>90</td>
<td>2</td>
<td>101</td>
<td>6</td>
<td>1664</td>
</tr>
<tr>
<td>102</td>
<td>40</td>
<td>40</td>
<td>3</td>
<td>102</td>
<td>10</td>
<td>1999</td>
</tr>
</tbody>
</table>
Above observations, leads to study the problem of skyline computation in multiple tables when join operation works on them. We propose approach with the aim to combine state-of-the-art join methods into skyline computation for any single join operation.

Nest-loops join also called nested iteration, is the simplest join method. Execution of NLJ is drawn directly from definition of Join [13]. The nested block join algorithm [13] is a more efficient version of the nested loop join algorithm. Block-oriented implementation of the nested-loops join optimizes on I/O overhead. Our implementation of joined skyline uses Nested block join to combine with skyline as it is supported by the entire database relational engine.

II RELATED WORK

Previous studies have introduced a host of different techniques for computing the skyline of a single relation [1, 2, 3, 5, 15]. Techniques that require an index over the input relation (e.g.,[1,15]) will need the join results to be computed in full and indexed before computing the skyline. Techniques that do not require an index (e.g.,[1,2]) can be pipelined with the join operator, but they are decoupled from the semantics of the join and essentially examine the complete join stream before terminating. Jin et al. [10] propose a multi-relational skyline operator that combines a sort-merge join algorithm with skyline processing. Sun et al. [14] propose a distributed adaptation of SaLSa [1,2] to compute the join skyline in a distributed setting. In a follow-up study of [10], two nonblocking algorithms for skyline join are proposed [11]. In [12], the ProgXe framework is introduced for skyline join computation that supports progressive result generation. ProgXe partitions the input relations by using a multidimensional grid access method. Finally, the particular access method is not part of conventional database systems and hence requires an additional implementation effort. The techniques that we develop rely simply on sorted access, which is implementation with existing relational operators, and do not require tuning.

III PRELIMINARY

This section presents our approach of computing skyline within join algorithm. To introduce our approach, we recall the dominance, and join attributes definition first and also formally present the problem considered.

3.1 Dominance

Let R be a relation with n-dimensional space D = (D1, ..., Dn) where dimensions D1, ..., Dn are in the domain of numbers. For any p > q, q is called a dominated object, and denote the set of objects dominating q as Dom(q).

3.2 Join Attributes and Descriptive Attributes

Given two relations R1 (d1, ..., dn, d1’s1, ..., dn’sn) and R2(d1, ..., d1’s1, ..., dn’sn) where attributes d1, ..., dn in R1 and d1’s1, ..., dn’sn in R2 are called join attributes such that they are only used in the join operation. Rest of the attributes not participating as join attributes are descriptive attributes. Descriptive attributes participate in the evaluation of skylines.

3.3 Skyline Join Problem

Given two relations R1 (d1, ..., dn, d1’s1, ..., dn’sn) and R2(d1’s1, ..., dn’sn) find skylines over the joined table R1 ⊗ R2 excluding the join attributes. We assume the join operator to be non-reductive and has the following properties:

1. The predicate is of the form x = y, where x is an expression computable from one table and y is an expression involving the other table;
2. It can be inferred that i) x cannot be null, and ii) for each x there must exist at least one y such that x = y holds.

3.4 Working of BNLSJ

It is discussed and proved in [5] if tuples are sorted in ascending order on Entropy any tuple r cannot be dominated by tuples following it. The same can be applied when evaluating skyline in the presence of join, if the joined tuples are produced with sorted entropy values, the skyline results among them can be found progressively i.e. that can quickly return the initial results without reading the entire database. Sorting on Entropy values, in a joined relation is difficult to accomplish. We sort R and S on entropy values individually, apply the Block Nested Loop Join algorithm to the sorted R and S, and then pipeline the joined results to SFS [5]. As SFS is progressive in behavior we claim our BNLSJ produce progressive tuples.

IV THE BLOCK NESTED LOOP JOIN SKYLINE (BNLJS)

4.1 The Algorithm

Input: (a) Relation R and S
(b) Join attributes r’s in R and s’s S.
Output: Skyline in R1⋈R2

Process:
1. Jointuple = Empty; // Initialization.
2. JoinSkyline = Empty; // Initialization.
   //Join loop execution/
4. For each block of R, in B[j],
   [B[j] is number of block used for Join ]
5. For each Page in S do
6. For all InMemory tuple r ∈ R and s ∈ S
7. If r = s Then Merge (Jointuple , r ⊙ s) ;
8. If Jointuple is Full PGS (Jointuple)
9. End If;
10. End For;
11. End For;
12. Return Skyline R⋈S = SFS (Jointuple).

4.2 Progressive Behavior of BNLJS

Let R and S are sorted w.r.t entropy values.

Sorted Tuples in R are r1, r2,.................... rR and S are s1, s2,............. sS.

As discussed and proved in [5]

For any tuple r, since Ent(r) ≤ Ent(r) (i < i'), r cannot be dominated by r
Similarly, any tuple s cannot be dominated by tuple s (j < j).
For any r, r (i < i') and s, s (j < j).

We study the same for Join tuples on R ⊙ S.

Let Ent(r) < Ent(r)

Case 1. If r joins s and r joins s, r ⊙ s cannot be dominated by r ⊙ s.

Case 2. If r joins s and r joins s, r ⊙ s cannot be dominated by r ⊙ s either although Ent(r ⊙ s) might be larger than Ent(r ⊙ s).

Let Ent(r) = Ent(r)

Case 1. Their skyline attribute values are not equal; the joined tuples satisfy non dominated ordering property.

Case 2. If Ent(r) = Ent(r) and both have the same skyline attribute values, the join order of r ⊙ s and r ⊙ s follows non-dominated ordering.

If r ⊙ s is generated before r ⊙ s, the Merge() function will switch their order in the join results to keep this non dominated ordering property. Therefore skyline objects are found progressively.

4.3 Cost Estimation for proposed BNLJS algorithm

For B1 buffer pages, one page is used to read data from S, while B1/3 pages are used for reading blocks of data from R for joining. The remaining buffer is split into two parts: one part for storing joined results and one part for PGS to process the skyline evaluation.

The Cost of BNLJS = Cost of sorting + cost of skyline computation.

Cost of sorting

The I/O cost of sorting is:

Cost BNLJS = XSortR(R) + XSortS(S) + 3 · BNLJS(R, S).

The CPU cost of sorting is:

TBNLJS = XSortcpu(R) + XSortcpu(S) + Merge_cpu(R) + Merge_cpu(S) + C

The I/O cost of Join phase is:

Cost BNLJS = PGS(S, (R ⊙ S)sky) + \frac{\left | \left ( R \bowtie S \right )_{sky} \right |}{B_4}.

(dR + dS)

the CPU cost of Join phase is:

TBNLJS = PGS_cpu(nF) = (dR + dS) · % of tuples in R matching with tuples in S. average number of tuples of S that each matchable tuple of R can match · C

The total cost is = (Cost BNLJS1 + Cost BNLJS2) . (weight for a page access transform to time) + TBNLJS1 + TBNLJS2.

V Experimental Evaluation

We implemented the method in C++. The join is performed using list structure and quick sort is applied on sorting the tuples. All the experiments are performed on Pentium 4, 2.2 Ghz CPU with 2 GB main memory and 80 GB hard disk, running Microsoft Windows XP. The number of records in BNL join block is 500 and each attribute is of 4 bytes.

The synthetic dataset is generated with random dataset generator. Two relations R (Customer data) and S(Order data) of independent data created with a join attribute(Customer Number) and skyline comparison are performed in 3 dimension is used. For implementation simplicity we consider only attributes participating in skyline comparison. For the purpose of result comparison we implemented basic block nested loop on PGS for skyline evaluation. To compare results we
implemented the same with first Join the relations on BNLJ and above join the PGS is applied. We observed though performance, dimensionality and I/O cost comparisons are slightly improved our developed method shows drastic improvement in progressive behavior of results (figure 1). The results are compared on following criteria.

**Performance Test:**
Performance is tested with consideration; cardinality of $S$ is 5 times of cardinality of $R$. We very cardinality of $R$ to 20K, 40K, 60 K, 80 K and 100K. The performance results are shown in Table IV.

### Table IV

**Performance Test, NLJ based Skyline**

| $|S|$ | Time in Seconds |
|---|---|---|
| 100K | 7.2 | 6.0 |
| 200K | 36 | 31 |
| 300K | 50 | 46 |
| 400K | 78 | 72 |
| 500K | 87 | 81 |

**Effect of dimensionality:**
The Relation $R$ and $S$ with 1:5 cardinality and tuple of 20K and 100K are used with increasing skyline dimensionality. The Skyline Dimension we vary 3d,4d,5d,6d,7d. Time of skyline results in our algorithms are shown in Table 1.5

### Table V

**Effect of dimensionality NLJ based Skyline**

<table>
<thead>
<tr>
<th>Skyline dimension</th>
<th>Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Join</td>
<td>Basic NLJ</td>
</tr>
<tr>
<td>3d</td>
<td>7.2</td>
</tr>
<tr>
<td>4d</td>
<td>10.8</td>
</tr>
<tr>
<td>5d</td>
<td>13.6</td>
</tr>
<tr>
<td>6d</td>
<td>17</td>
</tr>
<tr>
<td>7d</td>
<td>54.2</td>
</tr>
</tbody>
</table>

**Input/output Cost:**
With each read /write to or from file with varying cardinality in counted using a couter .For testing same data set used in 5.9.1 the results are indicated in Table VI.

### Table VI

**I/O Performance, NLJ based Skyline**

| Cardanility $|S|= 5*|R|$ | I/O count |
|---|---|---|
| | Basic NLJ | BNLJS |
| 100K | 9723 | 8995 |
| 200 K | 17234 | 16754 |
| 300 K | 31200 | 23459 |
| 400 K | 72156 | 59107 |
| 500 K | 78434 | 62657 |

**Progressive Evaluation:**
Progressiveness of an algorithm is the time taken in producing first output. Idea behind the research is efficiency during skyline join. One of the drawback of naïve method is the results are blocked by execution. We claim our developed methods are progressive in behavior and tested on 100K and 500K cardinality of $R$ and $S$ the time to receive first 1 to 200 skyline outcome is measured. Table VII holds the resultant data.

### Table VII

**Progressive Evaluation, NLJ based Skyline**

<table>
<thead>
<tr>
<th>Produced Skyline points</th>
<th>Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic NLJ</td>
<td>BNLJS</td>
</tr>
<tr>
<td>1$^{st}$ Sky</td>
<td>17.3</td>
</tr>
<tr>
<td>50$^{th}$ Sky</td>
<td>32.5</td>
</tr>
<tr>
<td>100$^{th}$ Sky</td>
<td>39.3</td>
</tr>
<tr>
<td>150$^{th}$ Sky</td>
<td>46.7</td>
</tr>
<tr>
<td>200$^{th}$ Sky</td>
<td>52.1</td>
</tr>
</tbody>
</table>
VI CONCLUSION AND FUTURE WORK

We proposed methods of skyline combined with Join in this paper. The basic nested loop Join is already part of all data base system hence its adaptation and performance is obvious. We combined block nested loop join execution strategy with s Sort First skyline. Our results demonstrate progressive behavior of combined join and skyline in compare to implementing skyline as add on operator in skyline join query. We will be further implement Indexed Nested loop join in combining with Skyline evaluation. We expect if input tuples are indexed will reduce the sorting phase of BNLJS leads to better results.

REFERENCES


Figure 1: Progressive Behaviour of BNLJS over Naive method.