Skyline-Join in Distributed Databases

Dalie Sun #1, Sai Wu *2, Jianzhong Li #3, Anthony K. H. Tung *4

#*Department of Computer Science and Technology, Harbin Institute of Technology*
Harbin, China
[1]sdl@hit.edu.cn
[3]lijzh@hit.edu.cn

**School of Computing, National University of Singapore*
Singapore
[2]wusai@comp.nus.edu.sg
[4]atung@comp.nus.edu.sg

***Abstract*— The database research community has recently recognized the usefulness of skyline query. As an extension of existing database operator, the skyline query is valuable for multi-criteria decision making. However, current research tends to assume that the skyline operator is applied to one table which is not true for many applications on web databases. In web databases, tables are distributed in different sites, and a skyline query may involve attributes of multiple tables.**

In this paper, we address the problem of processing skyline queries on multiple tables in a distributed environment. We call the new operator skyline-join, as it is a hybrid of skyline and join operations. We propose two efficient approaches to process skyline-join queries which can significantly reduce the communication cost and processing time. Experiments are conducted and results show that our approaches are efficient for distributed skyline-join queries.

I. Introduction

Given a set of attributes of interest, a skyline query retrieves the tuples which cannot be dominated by others in any of the attributes. For example, to find a suitable accommodation, a student may propose a query "return the apartments which are both cheap and close to the school". Skyline queries are very useful in decision-making systems. Due to its importance, researchers have started to implement skyline operation in commercial DBMS (DataBase Management System) [1].

Existing research tends to assume that the skyline query is issued to a single table. That is, all the required attributes are from the same table. However, this assumption is no longer valid for the web environment, where data from multiple sources are required for query processing. For example, database *A* provides the airline ticket service, database *B* lists all information of hotels, and database *C* lists bus services. Suppose John wants to plan his holiday in Paris and has his holiday starting from May 11th. The query will be *"list me the airlines which take off on May 11th or May 12th with the lowest price, and cheapest 4-star hotels with room available within 3 hours of my arrival and from where I can take a bus with minimal stops to Louvre Museum"*. This query has the same characteristics of a skyline query, but it extracts data of several tables from different sites. In this paper, we call this type of query *skyline-join*.

To answer a skyline-join query, a naive approach is to join corresponding tables and then perform any existing skyline algorithm. But as shown in [2], the naive scheme incurs high overhead. Compared to the join result, the skyline result is much smaller in size. Most tuples will not appear in the final result and thus many join operations can be avoided. This problem is even more critical in distributed environment, where the tuples of tables must be transferred to other sites for processing. Logically, if we prune the redundant tuples before joining the table, we can achieve a great improvement in performance. Based on this intuition, we propose two algorithms to efficiently process skyline-join queries.

This paper makes the following contributions: First, it defines a new query operator: *skyline-join*. Second, it proposes two algorithms to process skyline-join. The first algorithm extends the SaLSa [3] algorithm to cope with multiple relations and the other one is an algorithm which prunes the search space iteratively. Finally, we show the performance of our scheme with extensive experiments.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 defines skyline-join operator and introduces the corresponding algorithms. Section 4 evaluates our scheme by extensive experiments. We conclude our paper in Section 5.

II. Related Work

Borzonyi et al. [4] first introduced the skyline operator into relational database systems, and three algorithms were proposed: the block nested loops (BNL), divide-and-conquer, and B-tree-based schemes.

To improve the performance of BNL, it was proposed in [3], [5] to sort tuples before applying BNL. The sorting function guarantees that once the processing terminates, the unread tuples are definitely not skyline points. In [3], various sorting functions are evaluated and it is found that the "maximum coordinate sort" achieves the best performance. Our extended SaLSa scheme also adopts this sorting function.

Some proposals exploit the index to facilitate skyline query processing. Two skyline algorithms based on bitmap and B^+-tree index are proposed in [6], where indices are used to prune the search space. In [7], [8], a spatial index in the form of an R^*-tree is used to prune the search space. In [7], nearest neighbor (NN) is used to prune the search space recursively.

978-1-4244-2162-6/08/$25.00 © 2008 IEEE

If we want to find the MIN skyline point, a query is issued to find NN of the origin, which is a skyline point definitely. This skyline point prunes the subspace dominated by it. And then another NN query is issued to find more results. Instead of processing NN queries one at a time, the algorithm in [8] tries to prune the R*-tree branches based on the best-first nearest neighbor algorithm [9].

Recently, processing skyline query in distributed environment has attracted much attention. It was proposed in [11] to extend the problem to super-peer networks, and a threshold based algorithm, SKYPEER, was developed to reduce the number of data transferred among the peers. In [12], Wang et. al. extended BATON [13] to support skyline queries. The multi-dimensional search space is partitioned adaptively and each subspace is assigned a unique *ID*, which is used to map to a peer node of BATON. The skyline queries are forwarded based on the routing and partition information.

The work that is most related to ours is that by Jin et al. [2]. Their method classifies tuples into three types: general skyline points (skyline points of the table), local skyline points (skyline points of the join group) and others. Tuples of the first two types are joined together to generate the final results, while the last kind of tuples are discarded for they are dominated by others. In our schemes, new criteria are proposed to further prune the search space.

III. Skyline-Join Algorithms

A. Problem Definition

For each table, tuple $t_i = (v_0, v_1, ..., v_d)$ dominates tuple $t_j = (v'_0, v'_1, ..., v'_d)$ if $v_i \geq v'_i$ for $0 \leq i \leq d$ and $\exists j (0 \leq j \leq d) \wedge (v_j > v'_j)$, where $\geq$ and $>$ define partial orders between values. The skyline operator retrieves the tuples, which cannot be dominated by the other tuples. Suppose v_i comes from more than one table, we then have a new problem – skyline queries against multiple tables. We introduce a new query operator, called *skyline-join* to process such queries.

We express the skyline-join query as $< S, C, J >$ where S represents skyline attribute set for computing the skyline, C represents a set of conditions given by the user to limit the tuples to be searched and J represents the join attributes. We will use *S(T), C(T)* and *J(T)* to represent the skyline attributes, condition attributes and join attributes of table T respectively.

For a user's query $< S, C, J >$, a skyline-join search space is defined as

Definition 3.1: **Skyline-join Search Space**
For a table set $\mathscr{T}$, T_{total} is the skyline-join search space for query $< S, C, J >$ i.f.f.
a) $T_{total} = T_0 \bowtie T_1 \bowtie ... \bowtie T_k, T_i \in \mathscr{T}$
b) $\forall att_i \in (S \cup C) \rightarrow \exists T_i (T_i \in T_{total}) \wedge (att_i \in T_i)$
c) We cannot find T'_{total}, which has the above two properties and less tables than T_{total}

Skyline-join queries should be processed in its search space. And the domination between different tuples in the search space is defined as following:

Definition 3.2: **Skyline-join Domination**
For a skyline-join query $< S, C, J >$ and its search space T_{total}, suppose $t_1 = (v_0, v_1, v_2, ..., v_d)$ and $t_2 = (v'_0, v'_1, v'_2, ..., v'_d)$ are two tuples of T_{total}. We say that t_1 dominates t_2 i.f.f.
$\forall att_i (att_i \in S) \rightarrow (v_i \geq v'_i) \wedge (\exists att_j (att_j \in S) \wedge (v_j > v'_j))$

For simplicity, we use $t_i \succ_{dom} t_j$ to represent that tuple t_i skyline-join dominates t_j and $t_i \succ_A t_j$ to represent that t_i skyline-join dominates t_j in attribute set A.

Definition 3.3: **Result of Skyline-join**
The result set R for a skyline-join query $Q =< S, C, J >$ in its search space T_{total} must have following properties:
a) $\forall t_i (t_i \in R) \rightarrow \neg \exists t_j ((t_j \in T_{total}) \wedge t_j \succ_{dom} t_i)$
b) Suppose $t_i \in R$ and $t_i = (v_0, v_1, ..., v_d)$, if $att_i \in C$, v_i must satisfy the corresponding conditions.

For ease of discussion, in the following sections, we assume the set of select condition $C = \emptyset$ and consider only numeric attributes. And we use *"MAX"* as the skyline condition in our discussion.

B. Modified Local Skyline

To efficiently process skyline-join operations, we modify the local skyline process accordingly. First, we group the tuples by the join attribute. Then we invoke a known skyline algorithm such as the one in [5] for each group to obtain local skyline points of the group. Finally, the general skyline points are computed from local ones. In addition, we generate a *temp* table to store the local skyline points, which is used in the further processing. As discussed in [2], tuples not in the *temp* table cannot generate any skyline-join results. In Algorithm 1, *getSkyline* can be any existing skyline algorithm. Note that the modified local skyline algorithm can successfully prune the dataset only if some group contains a large number of tuples. For this reason, the algorithm in [2] will not work well in the case when the join attribute has a small selectivity.

Algorithm 1 LocalSkyline(Table T, boolean f)
input: T is the table to be processed, f indicates whether to generate the temp table
output: result of the skyline points
1: group tuples of T by join attribute
2: **for** each group G_i **do**
3: LR_i=getSkyline(G_i)
4: **if** f **then**
5: write LR_i to temp table
6: $LR = LR_0 \cup LR_1 \cup ... \cup LR_k$
7: return getSkyline(LR)

C. Extend SaLSa to Multiple Relations

Another problem of the technique in [2] is the computation of dominators. Suppose T_1 and T_2 are two tables participating in the skyline-join operation. Let t_1 and t_2 be two tuples of T_1 and T_2 respectively. We use $dom(t_1)$ and $dom(t_2)$ to represent the tuple sets that dominate t_1 and t_2 respectively. $t_1 \bowtie t_2$ can produce skyline-join results only if

$$dom(t_1) \bowtie dom(t_2) = \emptyset$$

Algorithm 2 SaLSa(TableSet $\mathscr{T}$)

input: $\mathscr{T}$ are tables participating in skyline-join
output: result of the skyline-join results

```
1: for each table T_i ∈ 𝒯 do
2:     R_i = LocalSkyline(T_i, true)
3:     sort temp table T'_i by maximum coordinate value
4:     T_i = T'_i
5: join R_1, R_2,...,R_k to get partial result set RS
6: while true do
7:     get a join sequence JS from table T_1 to T_{k-1}
8:     reset P_stop
9:     for each tuple t_i ∈ T_k do
10:        if sum(JS, t_i)< P_stop then
11:            break;
12:        else
13:            if JS ⋈ t_i is not dominated by RS then
14:                add JS ⋈ t_i to RS
15:                update P_stop accordingly
16: return RS
```

In [2], this criteria is applied to prune the tuples before performing the join operation. However, computing $dom(t_i)$ for a tuple t_i is an expensive process, even if we use an index. Thus, our schemes try to prune the dataset by applying simple but efficient rules.

SaLSa [3] proposes a scheme, which first sorts the tuples according to some symmetric sorting function $\mathscr{F}$ and then prunes the dataset based on the function. More formally, let $D(\mathscr{F}, l)$ denote the unread domain at level l, we have

$$D(\mathscr{F}, l) = \{p_i \in D : \mathscr{F}(p_i) < l\}$$

We can stop the skyline processing once we detect that

$$\forall p_i \in D(\mathscr{F}, l) : p_{stop} \succ p_i$$

As mentioned earlier, we use the "maximum coordinate sort" as our sorting function and the approach of SaLSa can be extended to multiple relational case. For tables $T_1, T_2, ..., T_k$, we sort tuples of each table by maximum coordinate sorting function. Suppose t' is a random joining tuple of $T_1 \bowtie T_2 \bowtie ... \bowtie T_{k-1}$. Then, for $t_i \in T_k$, tuples $t' \bowtie t_1$, $t' \bowtie t_2$,..., $t' \bowtie t_k$ must be sorted on the maximum coordinate value as well. So we can apply SaLSa rule to this tuple group. Suppose t_{ij} is the ith tuple of table T_j. For a joining tuple $t_{a_1 1} \bowtie ... \bowtie t_{a_k k}$, let $M = max(g(t_{a_1 1}), ..., g(t_{a_k k}))$, where function g returns the maximum coordinate value of the tuple. P_{stop} is defined as the minimum M of current skyline-join results. Note that P_{stop} is updated as the new tuples are added to current skyline-join results. Algorithm 2 lists the detail of the scheme.

In line 2, we prune each join group by invoking Algorithm 1 and the local skyline points are stored in the temp table T'_i. Then, after joining the general skyline points of tables, we can get the partial results of skyline-join (line 5). From line 6 to 15, a modified SaLSa algorithm is applied to retrieve the rest of results. In line 7, we generate a partial join result for table T_1,...,T_{k-1}. And in the inner loop, the last table is scanned in descending order to finalize the skyline-join result. P_{stop} is updated as minimum M of current results and once the sum of all attributes in the next tuple is less than P_{stop} (line 10), we can stop the processing safely.

Table 1

A_1	A_2	A_3
80	60	1
70	90	2
30	60	2
50	50	3
40	20	4
30	20	4
20	40	5
40	40	6
50	30	7
5	10	8

Table 2

B_1	B_2	B_3
20	30	1
45	40	1
30	35	2
20	20	3
50	60	3
40	10	4
40	20	5
90	40	6
50	80	7
5	5	8

(a) Step 1

Table 3

A_1	A_2	A_3
30	60	2
50	50	3
20	40	5

Table 4

B_1	B_2	B_3
50	60	3
40	10	4
40	20	5

(b) Step 2

Fig. 1. Iterative Algorithm

D. Iterative Algorithm

The intuition of iterative algorithm comes from the following observation. Suppose we have two tables T_1 and T_2 and the skyline-join query is $Q =< T_1 \cup T_2, \emptyset, J >$. We can calculate the skyline results R_1 and R_2 for each table individually. To achieve the skyline-join result of the two tables, we perform two subset joins: $P_1 = R_1 \bowtie T_2$ and $P_2 = T_1 \bowtie R_2$. Tuples in P_1 and P_2 are partial results of skyline-join. The next step is to retrieve new results caused by join operation. Suppose the remaining result set is P_3, P_3 must have the following property:

1) $\forall t_i \in P_3 \rightarrow t_i \in T_1 \bowtie T_2$
2) $\forall t_i \in P_3 \rightarrow (\exists t_j \in P_1 \wedge t_i \succ_{S(T_2)} t_j) \vee (\exists t_j \in P_2 \wedge t_i \succ_{S(T_1)} t_j)$

Property 2 says that tuples in P_3 must dominate a tuple $t_j \in P_1$ for attributes of T_2 or a tuple $t_i \in P_2$ for attributes of T_1. The final skyline-join result is $P_1 \cup P_2 \cup P_3$.

Example 3.1: Given two tables in Figure 1(a) and a skyline-join query $Q =< (A_1, A_2, B_1, B_2), \emptyset, (A_3, B_3) >$, the iterative algorithm proceeds in the following way. First, we compute skyline results for each table individually. For table 1, its skyline point is $R_1 = \{(80, 60, 1), (70, 90, 2)\}$. And for table 2, its skyline point is $R_2 = \{(90, 40, 6), (50, 80, 7)\}$. We then join R_1 with table 2 and get results $P_1 = \{(80, 60, 1, 20, 30), (80, 60, 1, 45, 40), (70, 90, 2, 30, 35)\}$. (80,60,1,20,30) is removed from P_1 for it is dominated by the second result, (80,60,1,45,40). In the same way, we get results $P_2 = \{(40, 40, 6, 90, 40), (50, 30, 7, 50, 80)\}$. For the remaining tuples in table 2, if their B_1 value is larger than 30 (P_1's minimal value in B_1)or B_2 value is larger than 35 (P_1's minimal value in B_2), they may generate query results in P_3. Applying the same rule, we get two candidate tables in Figure 1(b). The pruned tuples are dominated by tuples in P_1 or P_2. We can safely remove them without missing any result. Now, P_3 is a subset of table 3 joining table 4. One iteration is completed. And we can start a new round.

In each iteration, we generate partial results of P_1 and P_2 and use them to prune the rest tuples. To avoid the case where P_1 and P_2 are empty, before performing the local skyline algorithm, we transfer the join attribute from one table to the other and remove the tuples that can not generate any join results. This is the same strategy as semi-join. To save network cost, the Bloom Filter [14] is adopted. We build a Bloom Filter for the join attribute of each table. We have

the following theorem:

Theorem 3.1: Let P be the partial result of skyline-join query Q=<S, C, J>. For a table $T \in S$ and $t_i \in T$, t_i can be pruned, if

$$\forall t_j \in P, t_j \succ_{S(T)} t_i$$

Proof: Suppose r is a join result of t_i. $\forall att_i \in S$, if $att_i \in T$, then $\forall t_j \in P \rightarrow t_j \succ_{\{att_i\}} t_i$. If $att_i \notin T$, then we can find $t_j \in P$ and $t_j \succeq_{\{att_i\}} r$. Otherwise, r must be a partial result of Q, e.g. r$\in P$, which contradicts with the assumption of theorem 3.1. ∎

Theorem 3.1 indicates that if a tuple is dominated by all the partial results, it can be safely pruned. In fact, Theorem 3.1 can not efficiently prune the tuples if some partial result has a very small value in an attribute. Suppose for attribute att_i, its data distribution is $P_i(x)$ (the probability of $att_i \leq x$) and its density function is $f_i(x)$. We assume that attributes are independent to each other. Let T_{kj}, S_k denote the size of table j and the size of skyline-join partial results in the kth iteration respectively.

Suppose A is the attribute set of table j involved in the skyline-join. In k^{th} iteration, after receiving a partial skyline-join result, the tuples that are dominated by all of partial results can be safely pruned. For a skyline attribute att_i, the expected minimal value of all partial results in this attribute is estimated as:

$$min(att_i) = \int_{0 \leq x \leq 1} f_i(x)(1 - P_i(x))^{S_k - 1} dx \quad (1)$$

So the table size after pruning is:

$$T_{kj} = T_{(k-1)j}(1 - \Pi_{att_i \in A} min(att_i)) \quad (2)$$

Equation 2 indicates that the theorem 3.1 cannot efficiently prune the data in each iteration. Hence, we introduce the concept of *skyline outsider*:

Definition 3.4: **Skyline Outsider**
Let s be a skyline point for attribute set A of table T. Its outsider is defined as those tuples of T that cannot be dominated by s in face of A.

Note that the outsider of a skyline point is not necessarily a skyline point as well because it may be dominated by other skyline points. For table A in Figure 1(a), there is no skyline outsider for any of the skyline points. But for table B in Figure 1(b), tuple $< 50, 60, 3 >$ is an outsider of skyline point $< 90, 40, 6 >$.

Theorem 3.2: Let P be the partial result of skyline-join query Q=<S, C, J>. Let O_j be the outsider set of skyline point t_j. For a table $T \in S$ and $t_i \in T$, t_i can be pruned, if
$\forall t_j(t_j \in P \wedge t_j \succ_{S(T)} t_i)$, t_i's join value does not refer to any outsider in O_j.

Proof: Because $t_j \succ_{S(T)} t_i$, if t_i is a final result of Q, t_i must dominate t_j in any attribute of $S - S(T)$, which indicates that t_i must join an outsider of t_j in other tables. However, as t_i's join attribute does not refer to any outsider of t_j. It can not be in the result set. ∎

A1	A2	A3
50	60	1
30	60	2
50	50	3
40	20	4

(a) table 5

B1	B2	B3
30	80	1
50	60	2
60	60	3
40	55	4

(b) table 6

Fig. 2. Example of Domination

When computing local skyline points, we also record outsiders of each skyline point. Let R be current skyline-join result. For a tuple $t \in T$, if $\exists t' \in R(t' \succ_{S(T)} t)$ and t's join value refers to some outsider of t', t is a promising result, otherwise t can be pruned. For example, tuple $< 20, 40, 5 >$ of table A in Figure 1(b) can be pruned, because it is dominated by the partial result $< 40, 40, 6, 90, 40 >$ in attributes of table A but its join value "5" does not refer to an outsider of this skyline point in table B of Figure 1(b).

Algorithm 3 IterativeAlgorithm(Table T_1, Table T_2)

input: T_1,T_2 tables participate in skyline-join
output: result of the skyline-join

1: $T_1' = T_1$
2: $T_2' = T_2$
3: **while** $T_1' \neq \emptyset$ and $T_2' \neq \emptyset$ **do**
4: prune T_1' and T_2' using Bloom Filter of join attribute
5: **if** first iteration **then**
6: R_1=LocalSkyline(T_1', true)
7: R_2=LocalSkyline(T_2', true)
8: T_1'=temp table of T_1'
9: T_2'=temp table of T_2'
10: **else**
11: R_1=skyline(T_1')
12: R_2=skyline(T_2')
13: P_1=$R_1 \bowtie T_2'$
14: P_2=$T_1' \bowtie R_2$
15: result=result+P_1+P_2
16: $T_1' = outsiderPrune(T_1')$,$T_2' = outsiderPrune(T_2')$
17: $T_1' = APDominatePrune(T_1')$,$T_2' = APDominatePrune(T_2')$
18: return result;

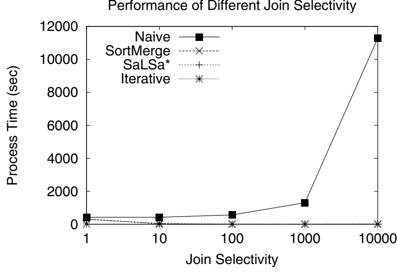
If tuple t is dominated by a partial result r and it passes r's outsider test, then t is inserted into a bucket of r. For k partial results, we have k buckets and each bucket contains some candidate tuples. Not all tuples of the bucket will appear in the final result.

Example 3.2: Suppose table 5 in Figure 2 represents the bucket for skyline point (30, 80, 1) of table 6. Other tuples in table 6 are outsiders of (30, 80, 1). A_1, A_2, B_1, B_2 are skyline attributes. A close observation reveals that (40, 20, 4) can be removed from the bucket for its join result (40, 20, 4, 40, 55) is dominated by the join result (50, 50, 3, 60, 60), produced by the tuple (50, 50, 3) in the same bucket.

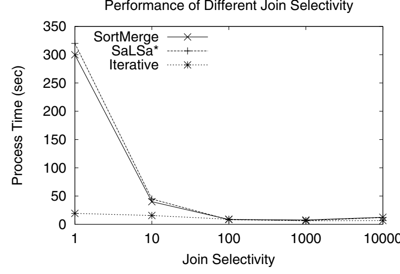
To decide whether a tuple in the bucket can be pruned, we should know:

1) The partial order sequence of tuples in the bucket.
2) The partial order sequence of tuples in other tables.

We cannot afford to compute the above information for the high overhead. Instead, an approximate approach is applied.



(a)



(b)

Fig. 3. Performance in Different Join Selectivity

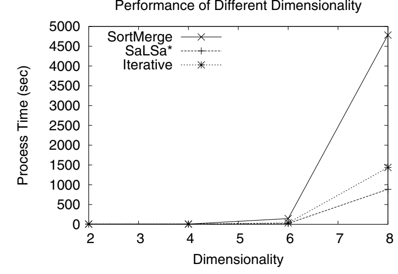
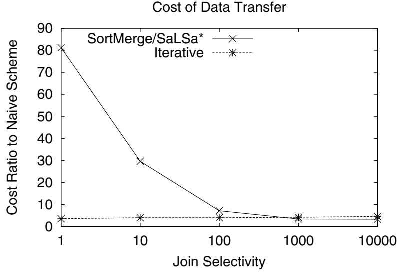
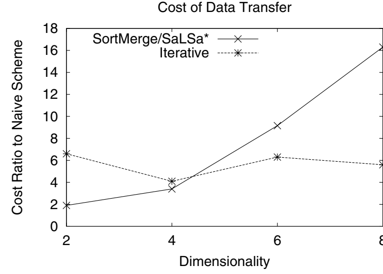


Fig. 4. Effect of Dimensionality



(a)



(b)

Fig. 5. Network Cost

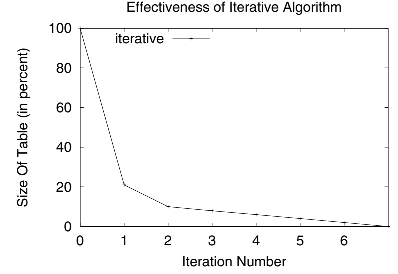


Fig. 6. Effectiveness of Pruning

For a tuple t_i of table T, we record its pruning power as:

$$\prod_{att_i \in S(T)} value(att_i)$$

This is the same strategy as [15]. As we group tuples by the join attribute, the pruning power of each group g is defined as the maximum pruning power of tuples in g. In addition, we should record the following information to efficiently prune tuples:

1) Let t_i be the tuple with maximum pruning power of g in table T, we record t_i's attribute values as $v(att_i), \forall att_i \in T$.
2) For each skyline attribute att_i, we record the maximum value $max(att_i)$ of the attribute appearing in g.

All the information can be obtained by a single table scan. Or we can record them when performing local skyline algorithm. To reduce network communication cost, we use Bloom Filter to record the above information. Suppose att_k's domain is *[low, up]*, we build K Bloom Filters for the attribute. The ith Bloom Filter responds for range $[\frac{i(up-low)}{K}, \frac{(i+1)(up-low)}{K}]$. For group g with join value j, to recognize its $max(att_i)$, we insert j into the Bloom Filter of att_i responding for the value $max(att_i)$. Other information is stored in the same way. Thus for table T, we need $S(T) * K$ Bloom Filters for recording tuples with maximum pruning power and additional $S(T) * K$ Bloom Filters for storing the maximum values of each joining group.

Finally, we define Approximate-Dominate as:

Definition 3.5: **AP-Dominate**
For a skyline-join query $Q =< S, C, J >$ and tuples t_i, t_j from the same table T, let g_i and g_j be the corresponding groups, tuple t_i is AP-Dominated by tuple t_j, i.f.f.
$(t_j \succ_{S(T)} t_i) \wedge (\forall att_k \in S \wedge att_k \notin S(T) \rightarrow g_j.v(att_k) > g_i.max(att_k))$

Theorem 3.3: For two tuples t_i and t_j of the same bucket, if t_i AP-Dominate t_j, t_j can be safely discarded.

Finally, the iterative algorithm is demonstrated in Algorithm 3. It first computes the local skyline points of each table or temp table. And then the partial result of skyline-join is generated and applied for pruning. The remaining tuples in the tables are pruned by outsiders and AP-domination. The performance of iterative algorithm relies on the number of skyline-join result, number of skyline attributes, data distribution, data correlation and accuracy of the Bloom filters. We delay the formal analysis to future work.

IV. Evaluation

In this section, we evaluate our schemes using two synthetic datasets. We use processing time as the major metric since skyline operation is CPU-intensive [4]. We use "sort-merge" to represent the method in [2], as this method is combined with sort-merge join. We also refer to the naive, extended SaLSa and iterative schemes as "Naive", "SaLSa*" and "Iterative" in our diagram respectively. By default, the join selectivity is set to 1.

A. Synthetic Dataset 1

Synthetic Dataset 1 is a series of independent datasets. Each dataset has 100K tuples. We also test our schemes on anti-correlated datasets. All the above schemes fail to prune the data space efficiently. Moreover, they introduce additional CPU cost due to the complex pruning process. In our experiment, the best method for processing anti-correlated datasets is to first invoke the modified skyline algorithm to generate a temp table of local skyline points and then applying naive scheme to find the skyline-join result. Due to space limitation, we omit the corresponding diagrams.

1) Effect of Join Selectivity: In this experiment, the total number of skyline attributes is 4. As shown in Figure 3(a), compared to other schemes, the naive scheme perform much worse especially when join selectivity is high. To better illustrate the performance of other schemes, we remove the line for naive scheme in Figure 3(b). The graph show that the iterative approach is more efficient and scalable.

2) Effect of Dimensionality: We also test the effect of dimensionality. Two independent datasets are generated. Each time, we pick 1, 2, 3 and 4 attributes of each table as the skyline attributes. Thus, the dimensionality of skyline-join is 2, 4, 6 and 8. We omit the naive scheme in the diagram for its worst performance. Figure 4 indicates that all schemes incur more overhead as the dimensionality increases. However, SaLSa* and iterative algorithms have better scalability than Sort-Merge. SaLSa* performs slightly better than iterative scheme when dimensionality is high.

3) Network Cost: We compare the network communication cost of each scheme for computing the skyline-join results. Naive approach transfers the whole table from one node to the other. Its network cost depends on the size of table. Sort-merge and SalSa* first prune the table by modified local skyline algorithm. Thus their costs rely on the join selectivity. Iterative algorithm applies partial results to progressively prune the dataset. So it achieves best results as showed in Figure 5. In Figure 5, we use Naive scheme as the baseline. All other schemes compute their percent of transferred data, compared to Naive scheme. The length of Bloom Filter is 1k bytes.

4) Effectiveness of Iterative Algorithm: The effectiveness of iterative algorithm is determined by its pruning ability. We show the table size after each iteration in Figure 6, which illustrates the case of 8 dimensions. The first iteration has the most powerful pruning effect. And in the following iterations, the table size decreases linearly. The major cost of iterative algorithm is produced by local skyline processing. In high dimensional case, local skyline processing incurs much more overhead. So if we can estimate the cost of next iteration and its benefit, we can decide whether to start a new iteration or just stop the algorithm and apply other schemes such as sort-merge and SaLSa*. We plan to set up an accurate estimation model in our future work.

B. Synthetic Dataset 2

We use tools offered by [16] to generate TPC-DS dataset. TPC-DS dataset is used as a benchmark for decision support. The join selectivity of TPC-DS tables can not be tuned, so we change the table size instead. We also do the experiments for dimensionality, network cost and effectiveness of iterative algorithm. The same results are returned as in synthetic dataset 1 and for space limitation, the diagrams are omitted.

In this experiment, we want to answer the following query:

```
SELECT * FROM Web_Sales, Item
WHERE ws_item_sk = i_item_sk
SKYLINE OF ws_quantity MAX, ws_net_profit MAX
        i_current_price MAX, i_wholesale_cost MAX;
```

Different size of table Web_Sales are generated to control the join result size. In Figure 7, iterative scheme outperforms the other two. And its performance is barely affected by the join result size.

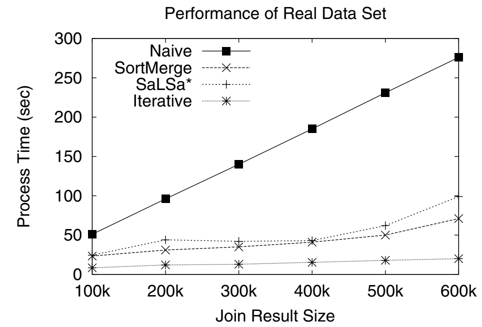


Fig. 7. Performance in TPC-DS

V. Conclusion

In this paper, we study the problem of skyline processing in multiple relations. We call this kind of operations skyline-join. Two schemes are proposed to handle skyline-join operations. One is the extension of existing skyline algorithm, SaLSa [3]. And the other one is an innovative iterative algorithm. We compare our schemes with the one proposed in [2]. The experiments indicate that iterative algorithm is more efficient and scalable. It also incurs less network communication cost in distributed environment. In our future work, we plan to set up an estimation model to combine iterative algorithm and other algorithms adaptively.

References

[1] S. Chaudhuri, N. Dalvi, and R. Kaushik, "Robust cardinality and cost estimation for skyline operator," in *ICDE*, 2006.

[2] W. Jin, M. Ester, Z. Hu, and J. Han, "The multi-relational skyline operator," in *ICDE*, 2007.

[3] I. Bartolini, P. Ciaccia, and M. Patella, "Salsa: Computing the skyline without scanning the whole sky," in *CIKM*, 2006.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001.

[5] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *ICDE*, 2003.

[6] K. L. Tan, P. K. Eng, and B. C. Ooi, "Efficient progressive skyline computation," in *VLDB*, 2001.

[7] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: an online algorithm for skyline queries," in *VLDB*, 2002.

[8] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An optimal and progressive algorithm for skyline queries," in *SIGMOD*, 2003.

[9] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," in *TODS*, 1999.

[10] J. Pei, Y. Yuan, X. Lin, W. Jin, M. Ester, Q. Liu, W. Wang, Y. Tao, J. X. Yu, and Q. Zhang, "Towards multidimensional subspace skyline analysis," *ACM Trans. Database Syst.*, 2006.

[11] A. Vlachou, C. Doulkeridis, M. Vazirgiannis, and Y. Kotidis, "Skypeer: Efficient subspace skyline computation over distributed data," in *ICDE*, 2007.

[12] S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu, "Efficient skyline query processing on peer-to-peer networks," in *ICDE*, 2007.

[13] H.V.Jagadish, B. C. Ooi, and Q. H. Vu, "Baton: A balanced tree structure for peer-to-peer networks," in *VLDB*, 2005.

[14] B. Andrei and M. Michael, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2003.

[15] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi, "Skyline queries against mobile lightweight devices in manets," in *ICDE*, 2006.

[16] "http://www.tpc.org/tpcds/tpcds.asp."