Skyline-Sensitive Joins with LR-Pruning*

Mithila Nagendra
CIDSE, Arizona State University
Tempe, AZ 85287-8809
mnagendra@asu.edu

K. Selçuk Candan
CIDSE, Arizona State University
Tempe, AZ 85287-8809
candan@asu.edu

ABSTRACT

Efficient processing of skyline queries has been an area of growing interest. Most existing techniques assume that the skyline query is applied to a single data table. Unfortunately, this is not true in many applications where, due to the complexity of the schema, the skyline query may involve attributes belonging to multiple tables. Recently, various hybrid *skyline-join* algorithms have been proposed. However, the current proposals suffer from several drawbacks: they often need to scan the input tables exhaustively in order to obtain the set of skyline-join results; moreover, the pruning techniques employed to eliminate the tuples are largely based on expensive pairwise tuple-to-tuple comparisons. In this paper, we aim to address these shortcomings by proposing two novel skyline-join algorithms, namely *skyline-sensitive join* ($\mathtt{S}^2\mathtt{J}$) and *symmetric skyline-sensitive join* ($\mathtt{S}^3\mathtt{J}$), to process skyline queries over multiple tables. Our approaches compute the results using a novel *layer/region pruning* technique (*LR-pruning*) that prunes the join space in blocks as opposed to individual data points, thereby avoiding excessive pairwise point-to-point dominance checks. Furthermore, the $\mathtt{S}^3\mathtt{J}$ algorithm utilizes an early stopping condition in order to successfully compute the skyline results by accessing only a subset of the input tables. We report extensive experimental results that confirm the advantages of the proposed algorithms over the state-of-the-art skyline-join techniques.

Categories and Subject Descriptors

H.2.4 [**Databases**]: Systems

General Terms

Algorithms, Performance

Keywords

Skyline-aware join processing

*This work is supported by NSF grant #1116394 "RanKloud: Data Partitioning and Resource Allocation Strategies for Scalable Multimedia and Social Media Analysis"

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.
EDBT 2012, March 26–30, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-0790-1/12/03 ...$10.00.

1. INTRODUCTION

Recently, there has been a growing interest in the efficient processing of skyline queries [4, 21, 22]. The *skyline* of a dataset is the subset of data points that are not dominated[1] by any other data points [4]; skyline queries are valuable in many multi-criteria decision support applications [1].

Various algorithms [2, 12, 20] have been developed to address the problem of discovering skylines for a wide range of scenarios. Most early algorithms assume that the skyline query is applied on a single table. Naturally, this assumption is not true for many applications where the schema is complex and the data is distributed onto many tables or data sources. This is especially true for information integration applications that inherently operate on multiple sources.

A naive approach to processing such queries is to first join the relevant tables to materialize all candidate tuples, and then, to apply existing single-table skyline algorithms. However, in most cases, a large percentage of the materialized tuples will not appear in the final skyline, and thus, a significant portion of the join work done to obtain the combined candidate set will be wasted. If, on the other hand, we could prune the redundant tuples during the join processing (i.e., the join operation is *skyline-sensitive*), then we could achieve significant improvements in the performance of multi-table skyline queries.

Recently, various attempts have been made to tackle this challenge [9, 18, 23, 20]. For instance, the `skyline-join` operator proposed in [20] is a hybrid of the previous skyline and join operators and leverages several pruning opportunities during its operation for faster execution. [23] proposes a non-iterative, single-pass `sort-first skyline-join (SFSJ)` operation for pruning tuples during the join. Nevertheless, like many others, this operator suffers from several drawbacks, including expensive tuple-to-tuple dominance checks. Motivated by this, we propose two non-iterative, single-pass skyline-join algorithms that avoid tuple-to-tuple dominance checks wherever possible;

- We develop a novel *skyline-sensitive join* ($\mathtt{S}^2\mathtt{J}$) algorithm that relies on a novel *layer/region pruning* (*LR-pruning*) strategy to avoid tuple-to-tuple dominance checks. $\mathtt{S}^2\mathtt{J}$'s key features are as follows:
  - The tuples in the outer table are organized into *layers* of dominance.
  - The inner table tuples are clustered into *regions* based on the Z-values of the skyline attributes.

[1]A point dominates another point if it is as good or better in all dimensions, and better in at least one dimension.

- A *trie*-based data structure on the inner table keeps track of the so-called *dominated*, *not-dominated*, and *partially-dominated* regions of the inner table *relative to the layers of the outer table*.
- S^2J obtains the skyline set by scanning the outer table, only once, while pruning the inner table.

- Next, we propose a *symmetric skyline-sensitive join* (S^3J) algorithm that repeatedly swaps the roles of the outer and inner data tables, and which rarely needs to scan any of the input tables entirely in order to obtain the set of skyline points.
- Experimental results show that the proposed algorithms are very efficient and outperform existing skyline-join algorithms on most datasets with different distributions and join selectivities.

The rest of the paper is structured as follows: in Section 2, we give an overview of the existing work. Section 3 presents the preliminaries and states the problem tackled in this paper. In Section 4, we discuss the proposed *skyline-sensitive join* algorithms in detail. Section 5 presents an extensive experimental evaluation of the proposed approaches. Finally, we conclude the paper in Section 6.

2. RELATED WORK

The task of finding the non-dominated set of data points was attempted by Kung *et al.* [11] in 1975 under the name of the maximum vector problem. Kung's algorithm lead to the development of various skyline algorithms designed for specific situations, including algorithms for high dimensional datasets [15] and parallel environments [19].

2.1 Skyline Algorithms

Borzsonyi *et al.* [4] were the first to coin and investigate the *skyline* computation problem in the context of databases. The authors extend Kung's divide and conquer algorithm so that it works well on large databases. They propose a *Block-Nested-Loops (BNL)* algorithm, which compares each point of the database with every other point and reports a point as a skyline result only if it is not dominated by any other point. They also propose a *divide and conquer* based algorithm, which divides the data space into several regions, calculates the skyline in each region, and produces the final skyline from the points in the regional skylines. By focusing on numerical domains, Borzsonyi *et al.* were able to gain logarithmic complexity along the lines of work done in [11].

The *Sort-Filter-Skyline (SFS)* algorithm [5], which is based on the same principle as the BNL algorithm, improves performance by first sorting the data according to a monotone function. Later contributions to skyline query execution include progressive (such as *bitmap* and *index* [21]) and online algorithms (such as [10] and [16] based on nearest-neighbor search). More recently, Huang *et al.* [6] proposed a parallel skyline algorithm for multi-processor clusters that utilizes Z-order clustering to reduce dominance checks.

2.2 Skylines on Multiple Tables

Some of the prior work on skylines over multiple tables include [23, 9, 18, 17, 8, 3, 20, 7, 13]. Jin *et al.* [7] integrate state-of-the-art join methods into skyline computation. Sun *et al.* [20] extend this work for distributed environments. The authors introduce a new operator, called `skyline-join`, and two algorithms to support skyline-join queries. The first extends the *Sort and Limit Skyline (SaLSa)* algorithm [2] to cope with multiple relations, whereas the second prunes the search space iteratively. These methods suffer from several drawbacks, including multiple passes over the datasets and complex book-keeping to identify pruned tuples.

In [8], the authors provide non-blocking methods for evaluating skylines in the presence of equi-joins. These algorithms are built on top of the traditional *Nested-Loop* and *Sort-Merge* join algorithms. Raghavan *et al.* in [17] propose a progressive query evaluation framework called *ProgXe* that transforms the execution of queries involving skyline over joins into a non-blocking form. They also enable skyline-join queries to be processed in a progressive manner, where the query processing (join, mapping, and skyline) is conducted at multiple levels of abstraction. *ProgXe* exploits knowledge gained from both input as well as mapped output spaces to enable executions of joins and skylines at a higher granularity of abstraction, rather than at the level of individual tuples. Following [17], Raghavan *et al.* in [18] propose a framework called *SKIN (SKyline INside Join)* to evaluate SkyMapJoin queries. SKIN reduces the total number of join results generated and the number of dominance comparisons needed to compute the skyline results by performing query evaluation at various levels of abstraction.

In [3], the authors develop various algorithms to efficiently process *aggregate skyline-join* queries. These algorithms locally process skylines as much as possible before carrying out the join between the tuples. [13] develops a framework called *FlexPref* for extensible preference evaluation in database systems. FlexPref aims to support a wide array of preference methods at the core of a database system. *PrefJoin* [9], which builds on FlexPref, is an efficient preference-aware join query operations designed specifically to deal with preference queries over multiple relations.

Most recently, Vlachou *et al.* [23] introduced the *Sort-First-Skyline-Join (SFSJ)* algorithm that fuses the identification of skyline tuples with the computation of the join. SFSJ computes the skyline set by accessing only a subset of the input tuples; it alternates between its inputs and generates the skyline tuples progressively as it computes the join results. The SFSJ algorithm relies on an early-termination condition, applied on a simple model of sorted input access, to determine whether it has accessed enough tuples to generate the complete skyline set.

The authors analyze the performance of SFSJ under 2 pulling strategies, simple round-robin (SFSJ-RR) and a novel adaptive strategy (SFSJ-SC), that define the order in which the input relations are accessed. The adaptive strategy prioritizes accesses to the input relations and is shown to be optimal for SFSJ in terms of the number of tuples accessed. SFSJ also provides a way to prune the input tuples if they do not contribute to the set of skyline-join results, thus reducing the number of generated join results and dominance checks. However, SFSJ does not perform the pruning in a block-based manner and largely depends on tuple-to-tuple comparison to find the region that is pruned. Our proposed approach aims to overcome this drawback by pruning the join space in terms of blocks as opposed to individual data points, thereby avoiding time-consuming pairwise point-to-point dominance checks.

In the following section, we formally introduce join-based skyline queries and provide a background to our approach.

3. PROBLEM DEFINITION

In this paper we focus on two-way join operations: Given (a) two datasets, $D_{out}(a_{out,1}, \ldots, a_{out,d_{out}})$ and $D_{in}(a_{in,1}, \ldots, a_{in,d_{in}})$, (b) a set of skyline attributes, $A_S \subseteq \{a_{out,1}, \ldots, a_{out,d_{out}}\} \cup \{a_{in,1}, \ldots, a_{in,d_{in}}\}$, and (c) a set of join attributes, $A_J \subseteq \{a_{out,1}, \ldots, a_{out,d_{out}}\} \cup \{a_{in,1}, \ldots, a_{in,d_{in}}\}$, a join-based skyline query seeks to identify a subset of $J = D_{out} \bowtie_{A_J} D_{in}$ consisting of tuples not *dominated* by any other tuples in J.

Let $p.a_h$ be the value of attribute a_h of tuple p; *p dominates q in the skyline attribute set, A_S, $(p \rhd_{A_S} q)$* when

$$\forall a_i \in A_S, (p.a_i \succeq q.a_i) \wedge (\exists a_k \in A_S \mid p.a_k \succ q.a_k).$$

Intuitively, p is better than or equal to $(\succeq)$ q in all dimensions in the skyline attribute set, A_S, and better than $(\succ)$ q in at least one dimension, a_k. In the rest of the paper, we omit the reference to the skyline attribute set and use $p \rhd q$ to denote that p *dominates* q (in the corresponding skyline attribute set). We use $p \not\rhd q$ for p *does not dominate* q.

DEFINITION 1. ***Join-based Skyline**. A data tuple, p, is in $D_{out}\ SSJ_{A_J,A_S}\ D_{in}$ iff (a) $p \in J = D_{out} \bowtie_{A_J} D_{in}$, and (b) $\nexists q \in J - \{p\}$ s.t. $(q \rhd_{A_S} p)$.*

Throughout the paper, join-based skyline operations are referred to as *skyline-sensitive join (SSJ)* operations. For example, given two tables `Highly_rated (rName, rating, location, remarks)` and `Business_hrs (rName, openingTime, closingTime)`, which contain information about restaurants, the query:

```
Skyline_results = SSJ Highly_rated by rName,
                      Business_hrs by rName,
                      rating MAX, closingTime MAX
```

would equi-join the tables on A_J = {Highly_rated.rName, Business_hrs.rName} and return results that are not dominated by any other results based on the skyline attributes A_S = {rating, closingTime}. In this query, the underlying preference function is MAX. While any monotonic function, such as MIN, is also acceptable as the preference function, in the rest of the paper, without loss of generality, we assume that MAX is specified by the user.

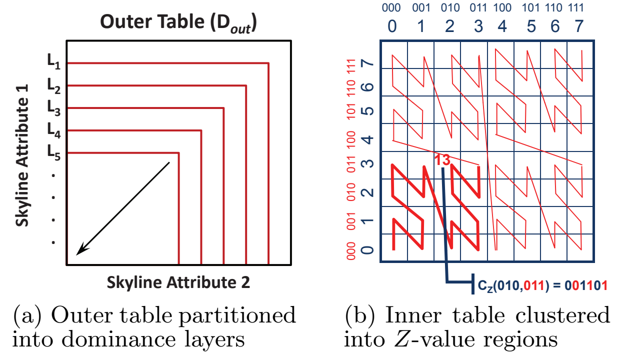
4. SKYLINE JOINS WITH LR-PRUNING

Skyline processing on a single table is an expensive operation as it may incur large access costs to perform pairwise dominance checks[2]. If a join operation is required to combine the relevant data, then the query will also incur additional costs for materializing the candidate tuples. Naturally, if the number of tuples materialized and compared can be kept low, significant improvements in the performance of skyline-join queries can be achieved.

Thus, we propose *skyline-sensitive join* algorithms where

- the data in the input tables are ordered in a manner that will help identify skyline points early and prevent unqualified data points from participating in the join operation; moreover,
- whenever possible, the join space needs to be pruned in terms of *blocks* as opposed to *individual* data points, therefore avoiding time-consuming pairwise point-to-point dominance checks.

[2]A dominance check compares two data points on a dominance condition.



(a) Outer table partitioned into dominance layers

(b) Inner table clustered into Z-value regions

Figure 1: Layer/region organization for S²J

We achieve these through a novel *layer/region pruning* (*LR-pruning*) strategy to minimize pairwise tuple comparisons: In LR-pruning, we partition the data in the outer table into *dominance layers*, while the inner table is clustered into *regions* using Z-order values of the skyline attributes to support block-based pruning. We then propose a second algorithm, *symmetric skyline-sensitive join* (S³J), that is similar to S²J in principle, but repeatedly swaps the roles of the outer and inner tables. One key outcome of this strategy is that (unlike S²J, where the outer table is fully scanned), in S³J, none of the datasets need to be scanned completely in order to obtain the skyline set.

Before we present S²J and S³J, we first provide an overview of the underlying LR-pruning strategy.

4.1 Layer/Region Organization of Data

4.1.1 Dominance Layering of the Outer Table

Let $A_{S_{out}} \subseteq A_S$ denote the skyline attributes that come from the outer data table, D_{out}. As shown in Figure 1(a), the outer table, D_{out}, is partitioned into dominance layers (using MAX). For a layer L_i, we define $max(L_i)$ as

$$max(L_i) = max(p.a | (a \in A_{S_{out}}) \wedge (p \in L_i)).$$

The dominance layers are obtained by sorting the tuples in the descending order of the value of $max(L_i)$. The algorithms exhaust all the tuples in the current layer before observing a decrease in the $max(L_i)$ value. In other words, the layers in D_{out} are such that for any entry in layer, L_i, the maximum of the skyline attribute values is larger than the maximum of the skyline attribute values for any entry in a dominated layer, L_h; i.e.,

$$\forall_{h>i, p \in L_i, q \in L_h} \quad max(p.a | a \in A_{S_{out}}) > max(q.a | a \in A_{S_{out}}).$$

Therefore, if two data points in D_{out} are such that $max(p.a | a \in A_{S_{out}}) = max(q.a | a \in A_{S_{out}})$, they will be grouped into the same layer partition.

The dominance layers are similar to the "bands" described in [23]. The layer partitions are ordered and accessed from the most dominant (L_1) to the least dominant (L_n) layer.

4.1.2 Region Organization of the Inner Table

Tuples in the inner table, D_{in}, are mapped onto a Z-order curve based on the corresponding skyline attribute set, $A_{S_{in}} \subseteq A_S$ (Figure 1(b)). The Z-order (or Morton-order [14]) curve, shown in Figure 1(b), is a fractal that can cover the entire data space by repeated applications of the same base pattern, a "Z". The Z-value of a data point

can be obtained by interleaving the bits of the binary representation of the coordinate values of the data point. In the example shown in Figure 1(b), the Z-value of $(2, 3)$ is 001101 obtained by interleaving the binary representations of 2 and 3, i.e. 010 and 011, respectively. For a d-dimensional space, in which the coordinate values fall in the range $[0, 2^v - 1]$, the Z-value of a data point contains $d \times v$ bits grouped into v number of d-bit groups. In the example, we have three 2-bit groups: 00, 11, and 01.

The Z-curve clusters neighboring points and regions in the space; points nearby in space also tend to be nearby on the curve. More specifically, given a Z-value with v many d-bit groups, the first bit group partitions the d-dimensional space into 2^d equi-sized regions (or clusters), the second bit group further partitions each of these 2^d region blocks into 2^d smaller equi-sized sub-regions (or sub-clusters), and so on. Also, when ordered by their Z-values, data points are naturally clustered into the regions of the space. For instance, in Figure 1(b), all the points located in the lower left quadrant of the space have prefix 00 in their binary representations and span Z-values from 0 to 13.

The Z-value mapping is one-to-one and invertible, i.e., the original coordinate values can be recovered from the Z-values. Moreover, given a prefix of the bit representation of a Z-value, we can determine the minimum ($minZ$) and maximum ($maxZ$) Z-values of that region. $minZ$ is obtained by filling the rest of the bit positions with 0s and $maxZ$ by setting all the missing bits to 1.

Note that, in the past, the clustering property of Z-curves has been leveraged for the single-table skyline problem [6, 12]. In this paper, we show that, *when used along with the dominance layering of the outer table* Z-values can be highly effective in eliminating redundant work for skyline joins.

4.2 S^{2}J Algorithm

Given two datasets, D_{out} and D_{in}, a set of join attributes, A_J, and a set of skyline attributes, A_S, the S^{2}J algorithm proceeds as follows:

1. $S_{all} = \emptyset$
2. S^{2}J scans outer table, D_{out}, from layer L_1 to L_n.
3. For each layer L_i:
   (a) S^{2}J invokes the `RegionsToExamine(T,` $max(L_i)$`)` function (see Section 4.2.2) to obtain the corresponding set of Z-value regions, R_i, from the trie structure, T, maintained for the inner table, D_{in}. The Z-value regions of the inner table obtained in this step will participate in the join-based skyline process with L_i.
   (b) $C = \emptyset$
   (c) For each region $r \in R_i$:
      - $C = C \cup (L_i \bowtie_{A_J} r)$ is carried out to combine the outer table tuples in L_i with the inner table tuples in r.
   (d) The skyline set, S_i, of $C \cup S_{all}$ is obtained.
   (e) `Rmarker(`S_i`)` is invoked to mark the appropriate regions of the inner table, D_{in}, based on S_i (see Section 4.2.1).
   (f) $S_{all} = S_{all} \cup S_i$
4. S^{2}J proceeds until all the layers in D_{out} are processed or the entire dataset D_{in} is pruned.

Algorithm 1: S^{2}J(D_{out}, D_{in}, A_S, A_J, T)

Input:
D_{out}: Outer table; D_{in}: Inner table; A_S: Set of skyline attributes of D_{out} and D_{in}; A_J: Join attribute set; T: Trie maintained for D_{in}; R: Set of regions to be examined during join-based skyline processing; $max(L_i)$: Maximum value of current layer in D_{out}

Output:
Skyline result S produced by D_{out} SSJ$_{A_J, A_S}$ D_{in}

Procedure:
```
Initialize T;
for each layer L_i ∈ D_out scanned from L_1, L_2, ..., L_n do
  if data point scanned is the first data point in L_1 or T has
  been changed then
    R = RegionsToExamine(T, max(L_i));
    /* Invoke the RegionsToExamine algorithm with parameters T
    and max(L_i) only if T has been changed, else R remains the
    same for the next round. RegionsToExamine is also invoked
    once at the beginning when the first data point in L_1 is
    scanned */
  end if
  initialize set S_i, the set of skyline points for L_i;
  /* Contains a set of points that do not dominate each other */
  for each r ∈ R do
    /* r is the Z-value region prefix */
    minZval(r) = minZ(r);
    maxZval(r) = maxZ(r);
    /* Minimum and maximum Z-values of r are obtained */
    joinSet = L_i ⋈_{A_J} D_in in the region defined by [minZval(r),
    maxZval(r)];
    add the results ∈ joinSet to S_i such that only skyline points
    are obtained in S_i;
    for each skyline point s = s_out||s_in ∈ S_i do
      U = min(s_out);
      T = Rmarker(s_in, T, U);
      /* Update T based on s_in ∈ D_in and the bound U ob-
      tained by taking the minimum of the coordinate values of
      s_out ∈ D_out */
    end for
  end for
  add S_i to S such that only skyline points are obtained in S;
end for
return S
```

Figure 2: The S^{2}J Algorithm

A more detailed pseudocode of S^{2}J is presented in Figure 2. In the following sections, we describe the details of the algorithm, and how region searches and markings are performed. The correctness of the S^{2}J algorithm is established in Section 4.2.4.

4.2.1 Region-based Pruning of the Inner Table

As it discovers new skyline points that can *prune regions* on the inner table, S^{2}J calls the `Rmarker` book-keeping function that marks regions of the inner table (Figure 3).

The `Rmarker` algorithm uses a *trie* data structure to efficiently store marked regions based on their common Z-value region prefixes. Each node of the trie holds one bit of the region prefix and a region marker that can be labeled as:

- *Not-Dominated* (denoted as ND) – this indicates that a inner table region is not (yet) dominated by any of the skyline points discovered so far.
- *Dominated for U* (denoted as DOM U) – this indicates that a region is dominated; U refers to the *largest* $max(L_i)$ of the outer table for which the region is guaranteed to be dominated. Intuitively, this marking keeps track of the layers of the outer table for which the region in the inner table can be considered as pruned.

Algorithm 2: Rmarker(s_{in}, T, U)

Input:
s_{in}: $s_{in} \in s = s_{out}||s_{in}$ – T marked using s_{in}; T: Trie structure of D_{in}; U: Bound on all the points that join s_{in} to form a part of the overall skyline result; q: Local Queue; l: Maximum depth of T

Output:
Updated Trie T

Procedure:
q.enqueue(1);
q.enqueue(0);
while q is not empty **do**
  $regionR$ = q.dequeue();
  if $s_{in} \rhd maxZ(regionR)$ **then**
    /* s_{in} dominates $regionR$ */
    if node n in T with prefix $regionR$ is marked as PD or ND or DOM U' such that $U > U'$ **then**
      mark n in T as DOM U;
    end if
  else if $s_{in} \not\rhd minZ(regionR)$ **then**
    /* s_{in} does not dominate $regionR$ */
    if node n with prefix $regionR$ is not marked as PD or DOM U'' and has no ancestor nodes marked as DOM U' **then**
      mark n in T as ND;
    end if
  else if $s_{in} \not\rhd maxZ(regionR)$ and $s_{in} \rhd minZ(regionR)$ **then**
    /* s_{in} may dominate some points in $regionR$ */
    $childRegion0$ = prefix $regionR$ appended with 0;
    $childRegion1$ = prefix $regionR$ appended with 1;
    if node n in T with prefix $regionR$ is a leaf node and length of prefix $regionR$ in not equal to l **then**
      /* new child nodes inserted into T, while limiting the depth of T to l */
      mark nodes with prefixes $childRegion0$ and $childRegion1$ with the region marker of their parent node n;
      /* Child nodes to be added to T inherit markers of parent node as default markers */
      if node n is marked as ND **then**
        mark n in T as PD;
        /* Parent node n, originally marked ND, is marked PD since its children are being explored further */
      end if
      insert nodes with prefixes $childRegion0$ and $childRegion1$ as children of parent node n in T
      q.enqueue($childRegion0$);
      q.enqueue($childRegion1$);
      /* Enqueue the child nodes of parent node n into q to be marked appropriately */
    else if n is an internal node **then**
      q.enqueue($childRegion0$);
      q.enqueue($childRegion1$);
      /* Enqueue the child nodes of parent node n into q to be marked appropriately */
    end if
  end if
end while
return T;

Figure 3: The `Rmarker` **Algorithm**

- *Partially-Dominated* (denoted as PD) – this indicates that a region itself is not dominated, but contains a subregion which is dominated.

The trie is initialized by marking the regions with prefixes 1 and 0 as *not-dominated* (ND) since the initial skyline set is $\emptyset$ (Figure 5(a)). As $\mathtt{S^2J}$ proceeds and new skyline points are found, the trie structure becomes deeper, capturing increasing amount of details. At each invocation of the `Rmarker` function, the nodes of the trie are considered from the root to the leaves. During this process, the markings of the internal nodes can be made *more specific* and the leaves of the trie may be split due to the newly discovered skyline points.

Let S_i be the skyline points set of layer L_i and let $s = s_{out}||s_{in} \in S_i$ be a skyline point; s_{out} corresponds to the outer table, whereas s_{in} corresponds to the inner table;

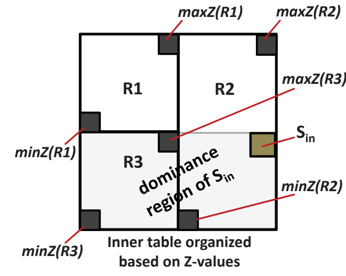
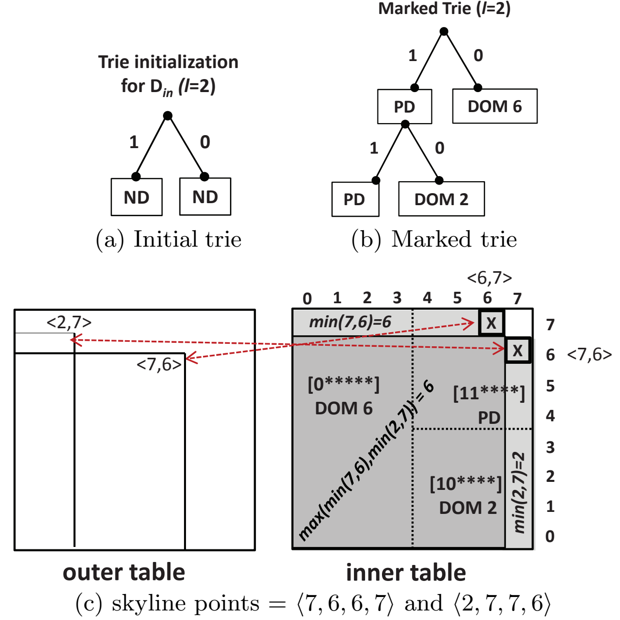


Figure 4: Z-value region based dominance test



(a) Initial trie (b) Marked trie

(c) skyline points = $\langle 7,6,6,7\rangle$ and $\langle 2,7,7,6\rangle$

Figure 5: (a) Initial trie; (b,c) the trie structure for skyline points $\langle 7,6,6,7\rangle$ and $\langle 2,7,7,6\rangle$; $s_{out} = \{\langle 7,6\rangle, \langle 2,7\rangle\}$ and $s_{in} = \{\langle 6,7\rangle, \langle 7,6\rangle\}$

- A node, η, that is originally marked as *Partially-Dominated* (PD) or *Not-Dominated* (ND) would become dominated if an s_{in} dominating this region is discovered. Let $S_{in}(\eta)$ be the set of skyline points on the inner table that dominates this node and $S_{out}(\eta)$ be the corresponding points on the outer table. Then, the node will be marked as DOM U, where U is

$$U = \max_{s_{out} \in S_{out}(\eta)} \left(\min_{a \in A_{S_{out}}} (s_{out}.a) \right).$$

Intuitively, U is the bound on the $max(L_i)$ of the layers of the outer table for which the region is guaranteed to be dominated. Consequently, this region need not be considered beyond any layer where $max(L_i) \leq U$.

- A node that is marked *Dominated* for U (DOM U) can be remarked as DOM U' if a bound $U' > U$ is found. Consequently, the algorithm progressively tightens the bound corresponding to a region, therefore pruning the region for an increasing number of outer table layers.

A leaf node may need to be expanded when a subregion of the node is found to be dominated by a new skyline point. If the leaf was originally marked *Not-Dominated* (ND), then it will now be marked as *Partially-Dominated* (PD) and two child nodes will be added to this node. If the leaf was marked *Dominated for U* (DOM U), it will be expanded *only if* the

Algorithm 3: RegionsToExamine(T, $max(L_i)$)

Input:
T: Trie structure of D_{in}; $max(L_i)$: Maximum value of current layer in D_{out}; s: Local Stack for DFS; l: Maximum depth of T

Output:
Set of regions R

Procedure:

```
s.push(0);
s.push(1);
while s is not empty do
    regionR = s.pop();
    parentMarker = current marker on node n having prefix
    regionR in T;
    if node n in T is an internal node then
        /* Node n has children */
        childRegion0 = prefix regionR appended with 0;
        childRegion1 = prefix regionR appended with 1;
        child0 = marker of node with prefix childRegion0 in T;
        child1 = marker of node with prefix childRegion1 in T;
        if parentMarker is DOM U such that U < max(L_i) then
            /* Finding a subregion in T marked DOM with the largest
            U < max(L_i) */
            if child0 is DOM U_0 such that U_0 > U and U_0 < max(L_i)
            then
                /* Explore the branches further */
                s.push(childRegion0);
                s.push(childRegion1);
            else if child1 is DOM U_1 such that U_1 > U and U_1 <
            max(L_i) then
                /* Explore the branches further */
                s.push(childRegion0);
                s.push(childRegion1);
            else
                add regionR to R;
            end if
        else if parentMarker is PD then
            /* Explore the branches further */
            s.push(childRegion0);
            s.push(childRegion1);
        end if
    else if parentMarker is ND or PD or DOM U then
        /* node n in T is a leaf node */
        add regionR string to R;
    end if
end while
return R
```

Figure 6: The `RegionsToExamine` Algorithm

subregion has an upper bound U' larger than U. When a leaf is expanded, the new child nodes are inserted into a queue and are examined and marked appropriately.

To control the granularity of the Z-value regions stored in the trie structure, we define a parameter l to constrain the depth of the trie: if a node is at the maximum defined depth l, then that node is not expanded any further. The effects of the depth maintained (l) in the trie structure on our skyline-join approach is experimentally analyzed in Section 5.2.

Similarly to [12], in order to decide whether a Z-value region is to be marked *Dominated for U* (DOM U), *Partially-Dominated* (PD) or *Not-Dominated* (ND), the `Rmarker` algorithm performs Z-value region based dominance tests. Let R be the prefix that represents a Z-value region; the region-based dominance tests apply the following three conditions:

1. If $s_{in} \rhd maxZ(R)$, then $s_{in} \rhd R$.
2. If $s_{in} \not\rhd maxZ(R) \wedge s_{in} \rhd minZ(R)$, then some points in region R may be dominated by s_{in}; hence this region needs to be explored further.
3. If $s_{in} \not\rhd minZ(R)$, then $s_{in} \not\rhd R$.

Figure 4 shows the region-based dominance tests of $R1$, $R2$, $R3$ against point s_{in}. Here, based on the above conditions, s_{in} dominates region $R3$ and does not dominate region $R1$. Region $R2$ needs to be explored further, since some of the points in this region may be dominated by s_{in}.

Figures 5(b) and 5(c) show the state of the trie ($l = 2$) after `Rmarker` is executed for the skyline points $\langle 7, 6, 6, 7\rangle$ and $\langle 2, 7, 7, 6\rangle$, where $\langle 7, 6, 6, 7\rangle$ is obtained by combining $\langle 7, 6\rangle$ from the outer table with $\langle 6, 7\rangle$ from the inner table and $\langle 2, 7, 7, 6\rangle$ is got by joining $\langle 2, 7\rangle$ with $\langle 7, 6\rangle$. Each of the trie nodes are labeled based on their overall coverage. For example, the node with prefix 10 is labeled DOM 2 because it is dominated by a skyline point and there is a subregion that is dominated only by $\langle 2, 7, 7, 6\rangle$ implying $U = min\langle 2, 7\rangle = 2$.

4.2.2 Fetching Non-Pruned Regions from the Trie

As the $\mathtt{S^2J}$ algorithm proceeds from one layer to another in the outer table, it calls the `RegionsToExamine` function to obtain the relevant regions from the trie structure (Figure 6).

The input to this function is the $max(L_i)$ value of the current outer table layer L_i. Given this value, `RegionsToExamine` returns the regions corresponding to the nodes of the trie that are currently marked ND and the regions related to the nodes that are marked DOM U with the largest $U < max(L_i)$. Note that, if a region corresponds to a leaf node of the trie and this node is currently marked PD, then this region is returned as well.

For example, given a level L_i with $max(L_i) = 7$, the trie in Figure 5(b) would return all leaf regions; in contrast, given a level L_i with $max(L_j) = 4$, it would only return the nodes with prefix 10 and 11. Note that, if a region r is marked *Dominated for U* and if $U \geq max(L_i)$, then this region does not participate in the join with the current layer L_i. This is because the layers are considered in the decreasing order of $max(L_i)$ and the bound U on a region monotonically increases; thus, once eliminated, a region will never be considered for any of the subsequent layers. This ensures that entire regions are pruned and the parts of the inner table that are pruned grow over time.

4.2.3 Layer-to-Region Joins

Once the non-pruned regions from the inner table are returned, $\mathtt{S^2J}$ joins the data in the current layer of the outer table with the data in these regions. To enable the join to be processed in an efficient manner, we use a B-tree index on the inner table, D_{in}, built using a composite key formed by the join attribute of D_{in} and the Z-values of the set of skyline attributes of D_{in}. Thus, given a join attribute value and a prefix of the relevant region, we can quickly identify matching skyline attribute values from the B-tree index: to join data tuples from the outer table layer L_i with the inner table region r, we first compute $minZ(r)$ and $maxZ(r)$ values of the region r (see Section 4.1.2) and then perform $(L_i \bowtie_{A_J} D_{in}[minZ(r), maxZ(r)])$ using the B-tree.

4.2.4 Soundness and Completeness of $\mathtt{S^2J}$

The $\mathtt{S^2J}$ algorithm (Figure 2) is correct in that, given two datasets, D_{out} and D_{in}, a set of join attributes, A_J, and a set of skyline attributes, A_S, the $\mathtt{S^2J}$ algorithm returns a set of results compatible with Definition 1 – i.e. it is sound (does not produce any non-skyline results) and complete (does not miss any skyline results).

Soundness: We first establish the soundness of $\mathtt{S^2J}$. Let $s = s_{out}||s_{in}$ be returned as a skyline point. Then, there is no $s' = s'_{out}||s'_{in}$ such that $s'_{out} \rhd s_{out}$ and $s'_{in} \rhd s_{in}$.

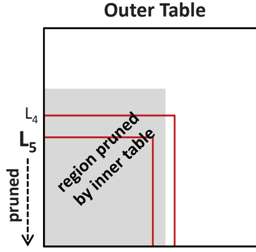


Figure 7: Scanning the (current) outer table stops at layer L_5 because L_5 and all the following layers are pruned by the (current) inner table

PROOF. Let us assume there exists such an s'. Note that s'_{out} will be in a layer, L', dominating or equal to the layer, L, of s_{out}; therefore, $max(L') \geq max(L)$.

- Since $s_{out}||s_{in}$ has been returned, when layer L has been considered, s_{in} was either not dominated or was dominated with a bound U where $max(L) > U$.
- Since $max(L') \geq max(L)$, s'_{out} is seen before s_{out}.
- Since when s_{out} was considered, s_{in} was either not dominated or was dominated with a bound U, when s'_{out} was considered earlier, s'_{in} (which dominates s_{in}) must also be either not dominated or dominated with a bound $U' \leq U$ (due to monotonicity of bounds).
- Since for s'_{out} we have $max(L') \geq max(L)$, where $max(L) > U$, and s'_{in} must be either not dominated or dominated with a bound $U' \leq U$, it follows that when s'_{out} is considered, s'_{in} has not yet been pruned.

Therefore, $s' = s'_{out}||s'_{in}$ which dominates $s = s_{out}||s_{in}$ would have been enumerated earlier than s, and thus, s' would have pruned, s contradicting the premise that s has been identified as a skyline point. Hence, there cannot be s' cannot be a skyline result. This proves that the S^{2}J algorithm is sound. □

Completeness: We next establish the completeness of S^{2}J. Let $t = t_{out}||t_{in}$ be a join result that has not been included in the skyline result. Thus, there must be an $s = s_{out}||s_{in}$, where $s_{out} \rhd t_{out}$ and $s_{in} \rhd t_{in}$, that is returned.

PROOF. Let us assume that there does not exist such an s. Let L be the layer in which t_{out} is considered.

- Since t has not been included in the result, when layer L is considered, t_{in} must be in a region dominated with a bound $U \geq max(L)$.
- Since t_{in} is in a region dominated with $U \geq max(L)$, there must have been an $s' = s'_{out}||s'_{in}$ where $s'_{in} \rhd t_{in}$ and $max(L') > max(L)$ (this is due to the monotonically decreasing property of $max(L)$ values).
- Since there does not exist $s \rhd t$ and since $s'_{in} \rhd t_{in}$, then $s'_{out} \not\rhd t_{out}$; i.e., $max(L') \leq max(L)$.

The last two statements above contradict each other; thus, there must exist an $s \rhd t$, and hence, t cannot be in the skyline. This proves the completeness of S^{2}J. □

4.2.5 *Complexity of* S^{2}J

The S^{2}J operator scans the outer table once and for each layer, L_i, it (a) identifies the relevant regions (that are either not dominated yet or dominated with a bound $U < max(L_i)$), (b) performs a join operation with these relevant regions of the inner table, and (c) finds the skyline based on the candidates.

Therefore, the overall cost is a function of the number of layers in the outer table, the size of the trie used for discovering the relevant regions, and the amount of pruning achieved based on the data distribution. In the extreme, a leaf stores an individual data point. Since this may impact the number of pairwise comparisons, in Section 5, we study the effect of the maximum allowed trie depth (l) on the performance of the S^{2}J algorithm.

Algorithm 4: S^{3}J(D_{out}, D_{in}, A_S, A_J)

Input:
D_{out}: Outer table; D_{in}: Inner table; A_S: Set of skyline attributes of D_{out} and D_{in}; A_J: Join attribute set; T_{in}: Trie maintained for D_{in}; T_{out}: Trie maintained for D_{out}; R: Set of regions to be examined during join-based skyline processing; $max(L_i)$: Maximum value of current layer in D_{out}

Output:
Skyline result S produced by D_{out} SSJ$_{A_J,A_S}$ D_{in}

Procedure:
```
Initialize T_in, T_out;
for each layer L_i^out ∈ D_out and L_i^in ∈ D_in do
  p(L_i^out) = ⟨max(L_i^out), max(L_i^out), ..., max(L_i^out)⟩
  find the region marker m_out for p(L_i^out) in T_out
  if m_out is ND or
  the largest bound U_max for m_out returned as DOM such that
  U_max < max(L_i^in) then
    S_i^out = S2J(L_i^out, D_in, A_S, A_J, T_in);
    /* S_i^out is skyline result set for L_i^out ∈ D_out returned by S2J
    */
  else
    stop scanning D_out
  end if
  p(L_i^in) = ⟨max(L_i^in), max(L_i^in), ..., max(L_i^in)⟩
  find the region marker m_in for p(L_i^in) in T_in
  if m_in is ND or
  the largest bound U_max for m_in returned as DOM such that
  U_max < max(L_i^out) then
    S_i^in = S2J(L_i^in, D_out, A_S, A_J, T_out)
    /* S_i^in is skyline result set for L_i^in ∈ D_in returned by S2J
    */
  else
    stop scanning D_in
  end if
  add Sout_i and Sin_i to S such that only skyline points are
  obtained in S;
end for
return S
```

Figure 8: S^{3}J swaps outer and inner tables in a round-robin manner

4.3 S^{3}J Algorithm

The S^{3}J algorithm (Figure 8) is similar to the S^{2}J, but swaps the roles of the outer and inner tables for each layer. Given two datasets, D_1 and D_2, a set of join attributes, A_J, and a set of skyline attributes, A_S, the S^{3}J algorithm maintains Z-value based indexes and trie structures for both D_1 and D_2. Each of the steps of S^{2}J is executed first assuming D_1 as the outer table and then D_2.

Since both tables are progressively pruned relative to each others' layers, S^{3}J algorithm may stop without fully scanning any of the tables (Figure 7). For this, S^{3}J utilizes an additional stopping condition not available to S^{2}J. Let D_{out} be the current table that serves as the outer table and D_{in} be the current inner table. Let L_i^{out} denote the layer for D_{out} and L_h^{in} denote the (recent) layer for D_{in}. When S^{3}J proceeds to the layer L_i^{out} in D_{out}, it first considers the extreme

point $p(L_i^{out}) = \langle max(L_i^{out}), max(L_i^{out}), \ldots, max(L_i^{out})\rangle$ of the layer and checks if the point is currently being dominated for any layers of D_{in} or not:

- If $p(L_i^{out})$ is not dominated, then the layer, L_i^{out}, is considered as before.
- If $p(L_i^{out})$ is dominated, then we find the largest bound U_{max} for which $p(L_i^{out})$ is dominated in the trie and check if $U_{max} \geq max(L_h^{in})$, where L_h^{in} is the current layer for the inner table, D_{in}. If so, then the data layer, L_i^{out}, and all the subsequent layers of the outer table have been eliminated from further consideration, and hence, the scanning of this table can be stopped.

The correctness of the early stop condition is established in the following section.

4.3.1 Correctness of S^3J

Soundness: Note that S^3J relies on the S^2J described in the previous subsection and the soundness of S^3J comes from the soundness of S^2J.

Completeness: Next, we prove that the early stop condition does not cause a violation of the completeness of S^3J. Let $t = t_{out}||t_{in}$ be a join result that has not been included in the skyline result. Since S^2J is complete, we know that if the layer containing t_{out} has been seen, then there must exist an $s \rhd t$ found earlier (see Section 4.2.4). We need to show that if t_{out} is not found due to early stopping, then there must also exist $s \rhd t$.

PROOF. Let L_{out} be the layer in which t_{out} would be considered if the early stop condition was not applied.

- Since the early stop condition has been applied and t_{out} has been pruned, we know that $p(L_{out})$ is dominated in the corresponding trie.
- Let U^{in} be the largest bound on the dominated regions covering $p(L_{out})$. Since the early stop condition has been applied and t_{out} has been pruned, we also know that $U^{in} > max(L_{in})$, where L_{in} is the current layer for the inner table.
- Since L_{in} is the current layer for the inner table, it includes t_{in}.
- Since

  $$U^{in} = \max_{s_{in} \in S_{in}} (\min_{a \in A_{S in}} (s_{in}.a))$$

  and since

  $$U^{in} \geq max(L_{in})$$

  it follows that there exits at least one $s_{in} \in S_{in}$ that dominates $p(L_{in})$, which also implies that $s_{in} \rhd t_{in}$.
- Since s_{in} is used in the computation of U^{in} on the region dominating t_{out}, there must be a corresponding $s_{out} \rhd t_{out}$.
- Therefore, there exists an $s = s_{out}||s_{in}$, where $s_{out} \rhd t_{out}$ and $s_{in} \rhd t_{in}$.

In other words, there exits $s \rhd t$ for any t that is eliminated from consideration due to the early stop condition. This proves that S^3J is complete. □

4.3.2 Discussion

Consider a scenario where we are given two tables that each contain data which are themselves skylines. On joining these tables (on a non-skyline attribute), each tuple in the join result will also be a global skyline point. This scenario constitutes the worst-case for our algorithm: since both the input data as well as the output consists of all skylines, there is no pruning or early stop possible. This implies that a full join of the input tables has to be performed (with the help of the index structure). In this scenario, the overhead of our algorithm is the amount of work needed to maintain the in-memory trie; which is simply $O(N \times l)$, where N is the amount of data and l is the depth of the trie (since none of the elements in the tables are pruned, our approach makes a root-to-leaf pass on the trie for each of the N data elements).

[23] has shown that the SFSJ algorithms are instance optimal with a ratio of 2 for cases where the inputs have at most K tuples in each "band" and have also shown that the algorithms are not instance optimal in the general case. As the experiments presented in the next section show, S^3J performs as good or better than SFSJ. Firstly, for each pruned region that SFSJ maintains, the S^3J algorithm maintains at least one marking in the trie. This implies that S^3J will stop as soon as (or earlier) than SFSJ. Moreover, S^3J is able to split a single region into multiple sub-regions that provide tighter boundaries. As a consequence, S^3J is able to make pruning decisions more proactively as opposed to SFSJ, which may need to see more data before it can stop.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the efficiency and effectiveness of S^2J and S^3J by varying the parameters involved and by comparing them against iterative skyline-join [20], PrefJoin [9], and the SFSJ-RR and SFSJ-SC algorithms [23].

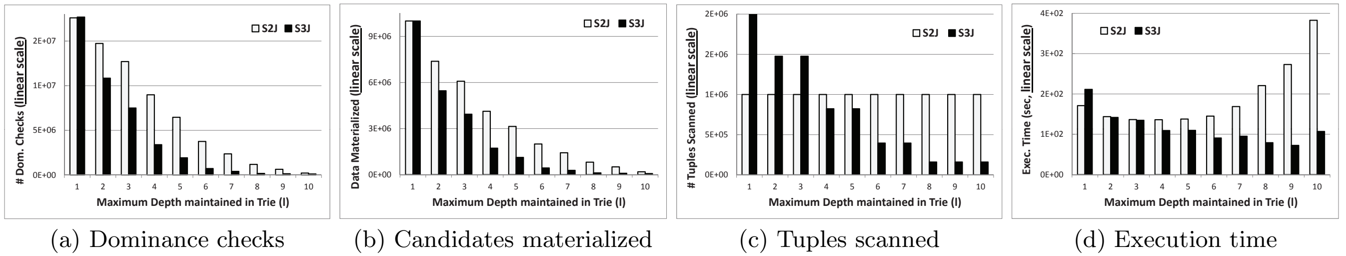
5.1 Experimental Setup

Our evaluations were conducted on a setup with an Intel Core 2 Duo 2.33GHz processor, 2GB RAM and Windows XP operating system. The S^2J and S^3J algorithms were implemented in Java and the experiments were run on data that was stored locally. For comparison purposes, the Java implementations of the SFSJ-RR and SFSJ-SC algorithms were obtained from the authors of [23]. In addition, we implemented PrefJoin [9] and the iterative skyline-join algorithm [20] to make further comparisons. The B-tree index is built using Berkeley DB Java Edition 3.3.87[3]. The trie implementation in Java was adapted from `http://www.technicalypto.com/2010/04/trie-in-java.html`.

- **Datasets.** Evaluations were carried out on 24 different synthetic datasets and 4 different real/benchmark datasets. Synthetic datasets as described in [4] were generated based on correlated, anti-correlated and independent distributions[4]. The cardinality of the datasets (n) was varied between 10,000 and 1 million tuples per data source, and the join rates (r) considered were 0.01 (i.e., 1 in 100 data tuples in a data source joins a tuple in another data source), 1, and 10. The dimensionality (d) of the skyline attribute set of each data source was varied between 2 and 4, hence

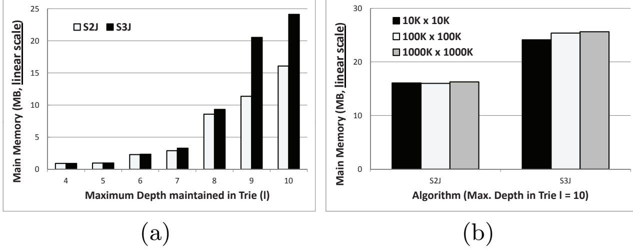
[3]Downloaded from `http://www.oracle.com/technology/software/products/berkeley-db/je/index.html`

[4]Random dataset generator available for download at `http://randdataset.projects.postgresql.org/`.



(a) Dominance checks (b) Candidates materialized (c) Tuples scanned (d) Execution time

Figure 9: Effect of maximum depth (l) of the trie structure ($n = 1M/dataset$, $j = 1$, $d = 2$, $s = 4$, $r = 10$)



(a) (b)

Figure 10: Main memory utilization by the trie: (a) Effect of maximum depth l ($n = 10K/dataset$, $j = 1$, $d = 2$, $s = 4$, $r = 10$), (b) Effect of cardinality n of datasets ($j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 10$)

the dimensionality of the result set ($s = |A_S|$) obtained after $D_{out}\ SSJ_{A_J,A_S}\ D_{in}$ varies between 4 and 8.

In addition to these synthetic datasets, we also used the NBA dataset[5] and the TPC-H benchmark datasets[6].

• **Evaluation Measures.** As is common in assessing skyline algorithms, we used execution time as the major metric in evaluating our methods. Execution time is the duration from the time an algorithm starts to the time it returns the entire skyline set. In addition, we used the total number of tuples scanned (sum of the depths of tuples accessed from the two input tables), the number of dominance checks and the number of join results as other evaluation metrics.

Both our approach and the competitor approaches require the inputs to be sorted. All indices and sorted records are prepared prior to running the experiments. Unless otherwise specified, each experiment is run five times and the results reported are the averages of the five runs.

5.2 Experimental Analysis of $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$

Before comparing $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ to other algorithms, we first experimentally analyze the performance characteristics of $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$. In particular, we investigate the impact of the region granularity defined by the trie depth l and the memory utilization of the data structures.

• **Impact of l.** If $l = 1$, the trie stores information about data points in only 2 blocks: one block that covers all data points with common region prefix 1 and another block that covers all data points with common region prefix 0. For a given l, the trie can store information up to 2^l regions.

Figure 9 shows the effect of l on execution time, number of tuples scanned, number of intermediate candidates materialized, and the number of dominance checks[7]. As can be seen here, for both $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$, the number of dominance checks (Figure 9(a)) and the number of intermediate data points materialized (Figure 9(b)) decrease as l increases. As expected, while for $\texttt{S}^2\texttt{J}$ the outer table needs to be scanned in its entirety independent of l, $\texttt{S}^3\texttt{J}$ is able to better prune the number of scanned tuples when the granularity of the space is fine; i.e., l is large (Figure 9(c)).

Figure 9(d) shows the execution time behaviors of $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$. As can be seen here, the impact of l on the execution time is not monotonic: as l increases, the execution times first decrease and, after some point, start increasing. Thus, there is a trade-off between the maximum depth of the trie and the gains achieved in terms of execution time.

Remember, from Section 4.2, that when the maximum depth of the trie is reached, the data in any "still non-dominated" regions need to go into a join operation with data from the other relation. Therefore, a higher trie depth may increase the chances of finding dominated regions that can then be pruned away. Therefore, higher values of l can help in achieving faster execution times. However, we observed that beyond a certain value of l it actually becomes cheaper to materialize the skyline candidates through joins, rather than repeatedly checking for possibly pruned regions within a dense trie structure. To see why, consider that in the worst case, for the maximum possible value of l, the trie stores the pruned data point itself at the leaf level. This would mean that the pruned regions would no longer be represented as blocks of data points, but instead they would be individual data points. Hence, checking for pruned regions will become as expensive, if not more expensive, as carrying out pairwise point-to-point dominance checks.

For the $\texttt{S}^2\texttt{J}$ algorithm the turning point comes early, between $l = 3$ and 5, whereas the $\texttt{S}^3\texttt{J}$ algorithm benefits from better granularity until $l \sim 9$. This is because $\texttt{S}^3\texttt{J}$ is able to stop without having to scan all the layers in the datasets.

• **Memory Utilization.** Our algorithms need 2 data structures: (a) a B-tree for indexing the data, and (b) a trie structure for maintaining the region markings.

The B-tree index, which is stored on disk and used for join processing, relies on a composite key formed by combining the join attribute and the Z-values of the set of skyline attributes (Section 4.2.3). While the use of the B-tree index itself is common (for index-nested loop joins) we have a slight increase in space overhead due to the Z-value composite keys needed for effective pruning. The overhead depends on the sizes of the Z-values relative to the original join keys. We ran an experiment on a dataset with 1 million tuples ($j = 1$, $d = 2$, $s = 4$, $r = 10$) and found that for this dataset the

[5]Downloaded from `http://skyline.dbai.tuwien.ac.at/datasets/nba/`.

[6]TPC-H database generator available at `http://www.tpc.org/tpch/default.asp`.

[7]Note that this experiment was carried on datasets with independent data distribution. Similar results were obtained for other datasets with different cardinality and dimensionality, and hence, these graphs are not shown.

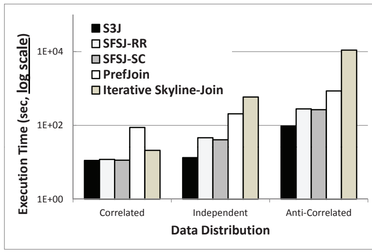


Figure 11: Comparison against other algorithms ($n = 100K/dataset$, $j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 5$)

Berkeley DB B-tree index built on only the join attribute occupied 40.3 MB of hard disk space, whereas an index built on the composite key formed by the join attribute and the Z-values occupied 96.9 MB of disk space. This rough doubling in space occupied is expected as the key and the Z-value each is stored as a Java integer of size 4 bytes.

Perhaps more important is the memory consumption of the in-memory trie data structure, which keeps track of the pruned and non-pruned regions. The space overhead of the trie depends on the number of marked regions; in the worst case, the trie would need to maintain markings on each and every data point; this would mean that the pruned regions can no longer be represented as blocks of data points, but instead they would be individual data points. As shown by the experimental results in Figure 10, the memory used by the trie structure mainly depends on the depth of the trie maintained (l), rather than on the cardinality (n) of the datasets. $\texttt{S}^3\texttt{J}$ uses more memory than $\texttt{S}^2\texttt{J}$, because $\texttt{S}^3\texttt{J}$ maintains a trie for both the outer and inner tables. Also, while the memory usage increases quickly with the depth of the trie, as the experiments in this section show, the overall depth that needs to be maintained is often not very high and the memory usage is negligible.

5.3 Evaluation against other Techniques

Based on the analysis in the previous subsection, we set $l = 5$ as the maximum allowed depth for the trie. Also, since the $\texttt{S}^3\texttt{J}$ algorithm performs better than $\texttt{S}^2\texttt{J}$, here we report comparison results only for $\texttt{S}^3\texttt{J}$.

In Figure 11, we first compare $\texttt{S}^3\texttt{J}$ against the iterative skyline-join [20], PrefJoin [9], and SFSJ methods [23] for a sample scenario. As can be seen here, $\texttt{S}^3\texttt{J}$ and the state-of-the-art approach, SFSJ, are the most competitive among the alternatives, and the difference from the other techniques is multiple orders of time. Therefore, in the rest of the section, we focus on the detailed study of $\texttt{S}^3\texttt{J}$ and SFSJ family.

5.3.1 TPC-H Benchmark and NBA Dataset

We first use the TPC-H benchmarks and the NBA dataset to compare the $\texttt{S}^3\texttt{J}$ algorithm to SFSJ-RR and SFSJ-SC .

• **TPC-H Datasets.** TPC-H is a decision support benchmark. The TPC-H datasets were generated using the TPC-H DBGEN tool[8]. We answered the following query:

```
Skyline_results = SSJ Part by P.partkey,
                  PartSup by PS.partkey,
                  P.size MAX, P.retailPrice MAX,
                  PS.availQty MAX, PS.supplyCost MAX
```

The cardinalities of the `Part (P)` and `PartSup (PS)` tables were varied by choosing different Scale Factors (SF). SF scales the database population – for a particular value of SF the DBGEN tool produces SF×200K tuples in table `Part` and SF×800K tuples in table `PartSup`.

Figure 12 illustrates that the $\texttt{S}^3\texttt{J}$ algorithm outperforms SFSJ-RR and SFSJ-SC on the TPC-H datasets over all evaluation metrics. Please note that all plots are on a log scale. In terms of execution time (Figure 12(a)), $\texttt{S}^3\texttt{J}$ shows significant gain as the size of the datasets increases.

As is expected, on extremely small TPC-H datasets (2K x 8K), $\texttt{S}^3\texttt{J}$ is marginally slower since it has the added overhead (~ 2 sec.) of accessing the trie structure to find the regions that have been pruned. But this overhead becomes negligible as the size of the dataset grows; this is mainly because $\texttt{S}^3\texttt{J}$ leverages the block-based LR-pruning technique to perform far lesser number of dominance checks (Figure 12(b)) as compared to the SFSJ methods. Both SFSJ-RR and SFSJ-SC use time-consuming pairwise tuple-to-tuple dominance checks in order to eagerly prune tuples that cannot produce skyline-join results, hence this causes the cost of finding the skyline to be considerably higher than our algorithm. In addition, $\texttt{S}^3\texttt{J}$ has a tighter stopping condition as compared to SFSJ-RR and SFSJ-SC (Figure 12(c)). It accesses lesser number of tuples than the SFSJ methods, hence showing that the early stopping condition employed by $\texttt{S}^3\texttt{J}$ is successfully able to avoid more number of redundant accesses to the tuples in the input datasets. Also, $\texttt{S}^3\texttt{J}$ materializes fewer number of intermediate skyline candidates (Figure 12(d)).

• **NBA Dataset.** For the experiments with the NBA dataset, we use the table that lists a player's regular season statistics – this table contains 21961 tuples and includes 17 types of statistics (e.g., assists and points) for the players. We encoded this dataset in the form of two tables: `Player_points (playerID, points, FGM)` and `Player_assists (playerID, assists, FTM)` – FGM stands for *Field Goals Made*, FTM stands for *Free Throws Made*. The following query was run:

```
Skyline_results = SSJ Player_points by playerID,
                  Player_assists by playerID,
                  points MAX, FGM MAX,
                  assists MAX, FTM MAX
```

As Figure 12 shows that the trends are similar to the results obtained using the TPC-H datasets, and the $\texttt{S}^3\texttt{J}$ algorithm outperforms the SFSJ-RR and SFSJ-SC algorithms on the NBA dataset as well.

5.3.2 Detailed Analysis using Synthetic Datasets

We now present a more detailed performance evaluation of the $\texttt{S}^3\texttt{J}$ algorithm using synthetic datasets.

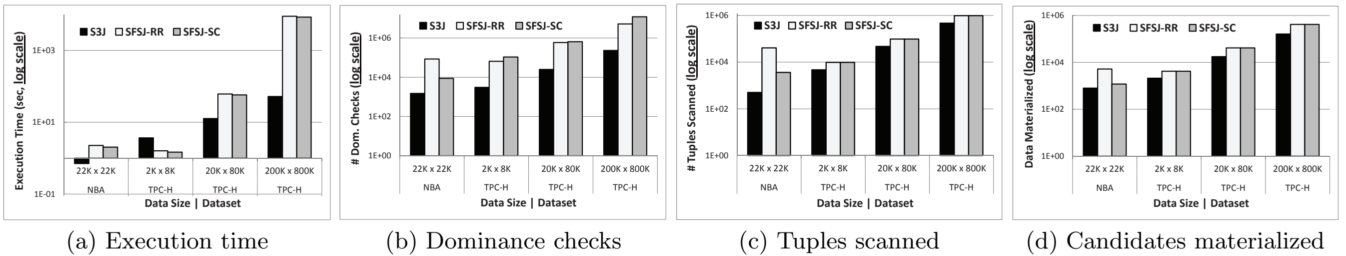
• **Effect of Data Correlation.** Figure 13 shows the performance of $\texttt{S}^3\texttt{J}$ over data with different correlations in terms of execution time (Figure 13(a)), dominance checks (Figure 13(b)) and total number of tuples scanned (Figure 13(c)). As illustrated in the figure, while $\texttt{S}^3\texttt{J}$ has only marginal gain on correlated datasets (in which only a few skyline points are obtained while the majority of the data points are dominated), it performs extremely well on independent and anti-correlated data distributions.

This shows that the LR-pruning technique and the early stopping conditions used in $\texttt{S}^3\texttt{J}$ are more effective than the corresponding pruning and stopping conditions in SFSJ, hence resulting in fewer number of dominance checks (Figure 13(b)) and lesser scans (Figure 13(c)).

In the rest of this section, we use independently distributed datasets in the interest of space.

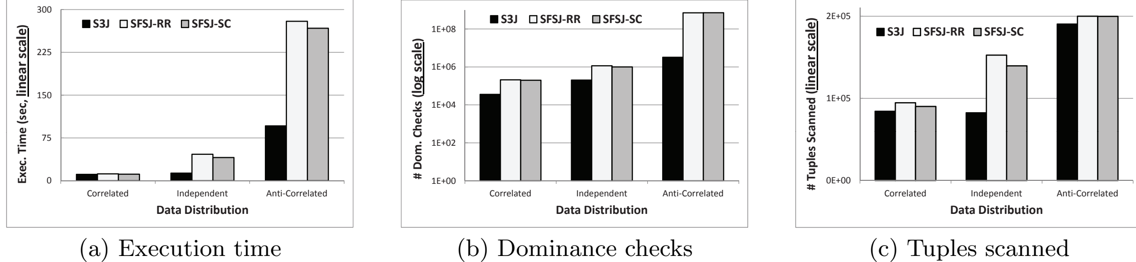
• **Effect of Dimensionality.** Figure 14 shows the impact of the number of skyline attributes per dataset ($d = 2, 3, 4$)

[8]See Footnote 6.



(a) Execution time (b) Dominance checks (c) Tuples scanned (d) Candidates materialized

Figure 12: Performance on real and benchmark datasets ($j = 1$, $d = 2$, $s = 4$, $l = 5$)



(a) Execution time (b) Dominance checks (c) Tuples scanned

Figure 13: Effect of data correlation ($n = 100K/dataset$, $j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 5$)

Outer Table Size	10K	100K	1000K
$\texttt{S}^2\texttt{J}$	4.11 sec.	14.82 sec.	137.87 sec.
$\texttt{S}^3\texttt{J}$	3.84 sec.	13.36 sec.	109.78 sec.
SFSJ-RR	2.49 sec.	46.31 sec.	9208.12 sec.
SFSJ-SC	2.47 sec.	40.59 sec.	7932.35 sec.

Table 1: Impact of size of the outer table on execution time ($j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 5$)

across the various evaluation metrics. The number of skyline attributes in each case is given by $s = 2 \times d$. As observed in the figures, the proposed algorithm outperforms both SFSJ-RR and SFSJ-SC in terms of execution time (Figure 14(a)), dominance checks (Figure 14(b)), and total number of tuples accessed (Figure 14(c)). The SFSJ methods fail to prune the space efficiently and incur additional cost due to its pruning process. This experiment illustrates that the proposed $\texttt{S}^3\texttt{J}$ algorithm is clearly more scalable as compared to SFSJ-RR and SFSJ-SC, and the trie-based LR-pruning technique is effective even on datasets with high dimensions. Similar results were obtained for other data distributions.

• **Effect of Data Cardinality.** $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ differ from each other in one key aspect. The $\texttt{S}^2\texttt{J}$ algorithm makes a distinction between the outer and inner tables. Table 1 shows the impact of the size of the outer table on $\texttt{S}^2\texttt{J}$ – as the size of the outer table increases the execution time increases as well. $\texttt{S}^3\texttt{J}$ improves on $\texttt{S}^2\texttt{J}$ by continuously swapping the roles of the outer and inner tables for each layer. The swapping of the tables eliminates the need to pick one of the tables as the "best" outer table. This hypothesis is confirmed by the experimental results obtained. For instance, on tables containing 1 million tuples each (Table 1), $\texttt{S}^2\texttt{J}$ has an execution time of $\sim$ 137 sec., whereas $\texttt{S}^3\texttt{J}$ has a faster execution time of $\sim$ 109 sec. Note that, $\texttt{S}^2\texttt{J}$ still performs better than SSFJ-RR and SFSJ-SC; on the above-mentioned dataset these methods have execution times of $\sim 9,208$ sec. and $\sim 7,932$ sec., respectively.

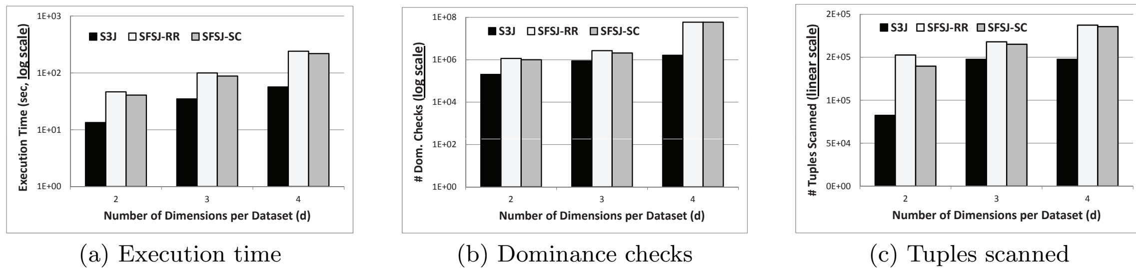
Figure 15 illustrates the performance of $\texttt{S}^3\texttt{J}$ against increases in data cardinality of each dataset (n = 10K, 100K, 1000K). The plots are on a log scale and show results on independently distributed datasets. Figure 15(a) shows that $\texttt{S}^3\texttt{J}$ scales well and performs significantly better in terms of execution time as compared to the SFSJ algorithms. On the very small dataset, $\texttt{S}^3\texttt{J}$ is marginally slower since it has the added overhead of accessing the trie structure in order to find the regions that have been pruned. But as the size of the datasets increases, $\texttt{S}^3\texttt{J}$ scales much better than SFSJ-RR and SFSJ-SC.

• **Effect of Join Rate.** Figure 16 compares the performance of $\texttt{S}^3\texttt{J}$ against the SFSJ algorithms across different join rates, namely $r = 0.01, 1, 10$. In terms of execution time (Figure 16(a)), the $\texttt{S}^3\texttt{J}$ algorithm outperforms both SFSJ-RR and SFSJ-SC across different joins rates. This gain is due to the fewer number of dominance checks (Figure 16(b)) performed by $\texttt{S}^3\texttt{J}$ and also the fact that it scans a lower number of tuples (Figure 16(c)) as it computes the skyline. The result shows that $\texttt{S}^3\texttt{J}$ has a clear advantage over the SFSJ methods even when the join rates of the datasets are varied.

6. CONCLUSIONS

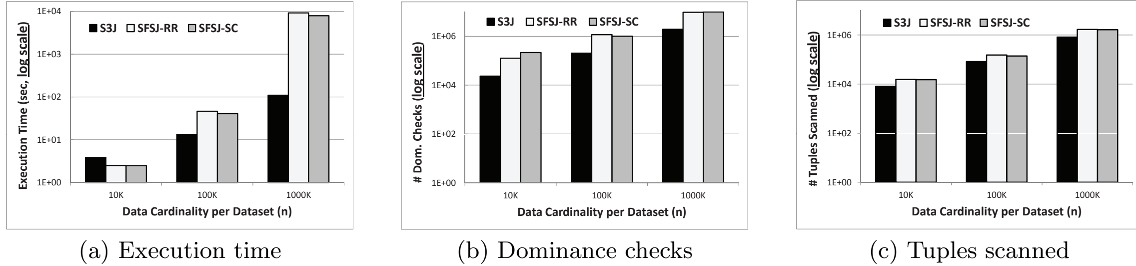
In this paper, we studied the problem of processing skyline queries over multiple data sources. We proposed two novel non-iterative algorithms, namely $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$, to process join-based skyline queries in a *skyline-sensitive* manner. Both of these algorithms produce the skyline points by scanning the outer table one dominance layer at a time and require *at most* a single scan. A *trie*-based book-keeping strategy helps prune the tuples in the inner table, which are mapped to their corresponding Z-order values, quickly. The Z-order values help cluster the data into region-blocks in order to support efficient dominance checks and to facilitate block-based pruning of the inner table. The $\texttt{S}^2\texttt{J}$ algorithm scans the outer table entirely, while pruning the inner table progressively. The $\texttt{S}^3\texttt{J}$ algorithm is similar to $\texttt{S}^2\texttt{J}$, the main difference being that $\texttt{S}^3\texttt{J}$ repeatedly swaps the outer and inner tables for symmetric operation. A special stopping condition applicable when using this symmetric strategy helps the algorithm stop earlier than $\texttt{S}^2\texttt{J}$, without having to scan any of the input datasets in its entirety.

The experiments carried out show the superiority in performance of $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ on both real and synthetic datasets



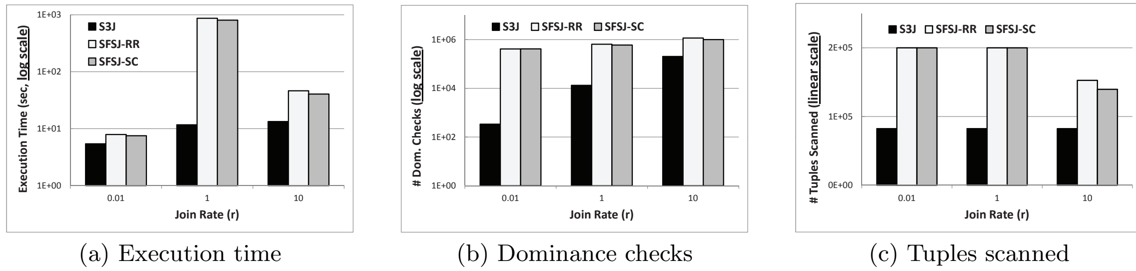
(a) Execution time (b) Dominance checks (c) Tuples scanned

Figure 14: Effect of dimensionality ($n = 100K/dataset$, $j = 1$, $s = 2d$, $r = 10$, $l = 5$)



(a) Execution time (b) Dominance checks (c) Tuples scanned

Figure 15: Effect of data cardinality ($j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 5$)



(a) Execution time (b) Dominance checks (c) Tuples scanned

Figure 16: Effect of join rate ($n = 100K/dataset$, $j = 1$, $d = 2$, $s = 4$, $l = 5$)

with various distributions, cardinalities and dimensions. In conclusion, the proposed algorithms are very efficient in executing join-based skyline queries and significantly outperform the existing skyline-sensitive join techniques.

7. REFERENCES

[1] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed Skylining for web info. systems. In *EDBT*, 2004.

[2] I. Bartolini, P. Ciaccia, and M. Patella. Salsa: computing the skyline without scanning the whole sky. In *CIKM*, 2006.

[3] A. Bhattacharya and B. P. Teja. Aggregate skyline join queries: Skylines with aggregate operations over multiple relations. In *COMAD*, 2010.

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

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

[6] J. Huang, J. Chen, Q. Du, and J. Yin. A load balancing skyline query algorithm in high bandwidth distributed systems. In *FSKD*, 2010.

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

[8] W. Jin, M. D. Morse, J. M. Patel, M. Ester, and Z. Hu. Evaluating skylines in the presence of equijoins. In *ICDE*, pages 249–260, 2010.

[9] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. Prefjoin: An efficient preference-aware join operator. In *ICDE*, pages 995–1006, 2011.

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

[11] H.-T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 1975.

[12] K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the skyline in Z order. In *VLDB*, 2007.

[13] J. J. Levandoski, M. F. Mokbel, and M. E. Khalefa. Flexpref: A framework for extensible preference evaluation in database systems. In *ICDE*, 2010.

[14] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, 1966.

[15] J. Matoušek. Computing dominances in E^n. *Information Processing Letters*, 38:277–278, 1991.

[16] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline computation in database systems. *ACM*, 2005.

[17] V. Raghavan and E. A. Rundensteiner. Progressive result generation for multi-criteria decision support queries. In *ICDE*, pages 733–744, 2010.

[18] V. Raghavan, E. A. Rundensteiner, and S. Srivastava. Skyline and mapping aware join query evaluation. *Inf. Syst.*, 36:917–936, September 2011.

[19] I. Stojmenović and M. Miyakawa. An optimal parallel algorithm for solving the maximal elements problem in the plane. *Parallel Computing*, 7:249–251, 1988.

[20] D. Sun, S. Wu, J. Li, and A. K. H. Tung. Skyline-join in distributed databases. In *ICDE Workshops*, 2008.

[21] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive Skyline computation. In *VLDB*, 2001.

[22] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel Skyline computation. In *SIGMOD*, 2008.

[23] A. Vlachou, C. Doulkeridis, and N. Polyzotis. Skyline query processing over joins. In *SIGMOD*, 2011.