

# Skyline Query Processing over Joins

Akrivi Vlachou<sup>1</sup>, Christos Doulkeridis<sup>1</sup>, Neoklis Polyzotis<sup>2</sup>

<sup>1</sup>Norwegian University of Science and Technology (NTNU), Trondheim, Norway

<sup>2</sup>UC Santa Cruz, California, USA

{vlachou,cdoulk}@idi.ntnu.no, alkis@cs.ucsc.edu

## ABSTRACT

This paper addresses the problem of efficiently computing the skyline set of a relational join. Existing techniques either require to access all tuples of the input relations or demand specialized multi-dimensional access methods to generate the skyline join result. To avoid these inefficiencies, we introduce the novel *SFSJ* algorithm that fuses the identification of skyline tuples with the computation of the join. *SFSJ* is able to compute the correct skyline set by accessing only a subset of the input tuples, i.e., it has the property of early termination. *SFSJ* employs standard access methods for reading the input tuples and is readily implementable in an existing database system. Moreover, it can be used in pipelined execution plans, as it generates the skyline tuples progressively. Additionally, we formally analyze the performance of *SFSJ* and propose a novel strategy for accessing the input tuples that is proven to be optimal for *SFSJ*. Finally, we present an extensive experimental study that validates the effectiveness of *SFSJ* and demonstrates its advantages over existing techniques.

## Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—Query processing

## General Terms

Algorithms, Experimentation, Performance

## Keywords

Skyline query, join operator

## 1. INTRODUCTION

In this paper, we propose a novel algorithm for efficiently computing the skyline set of a join  $R_1 \bowtie R_2$  without generating all the join tuples and without accessing all tuples of  $R_1$  and  $R_2$ . Our solution enables the progressive computation of the skyline join on large data sets or data derived from complex queries, and hence yields a very powerful tool for applications such as decision-making and complex data analysis (the typical use-cases for the skyline operator).

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.

*SIGMOD'11*, June 12–16, 2011, Athens, Greece.

Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...\$10.00.

HID	HName	Price	Rating	Location
$h_1$	Bridge Hotel	100	2	A
$h_2$	Tower Hotel	150	5	B
$h_3$	Hilton	200	10	A
$h_4$	Comfort Inn	400	8	A
$h_5$	Atlantis	300	3	C
$h_6$	Holiday Inn	350	7	B

RID	RName	Distance	Quality	Location
$r_1$	Egons	100	1	B
$r_2$	AB Bistro	250	3	C
$r_3$	Brewery	500	4	A
$r_4$	Joey's Bar	400	2	B
$r_5$	Cozy Lounge	200	1	C
$r_6$	DAlferno	500	1	A

HID	RID	HName	RName	Location	Price	Rating	Distance	Quality
$h_1$	$r_5$	Bridge Hotel	Brewery	A	100	2	500	4
$h_2$	$r_1$	Tower Hotel	Egons	B	150	5	100	1
$h_3$	$r_3$	Hilton	Brewery	A	200	10	500	4
$h_4$	$r_4$	Tower Hotel	Joey's Bar	B	150	5	400	2
$h_5$	$r_2$	Atlantis	AB Bistro	C	300	3	250	3
$h_6$	$r_1$	Holiday Inn	Egons	B	350	7	100	1
$h_6$	$r_4$	Holiday Inn	Joey's Bar	B	350	7	400	2

Figure 1: Example of skyline join.

Figure 1 illustrates an example of computing the skyline over the result of a join. Suppose that the database contains relations *Hotels* and *Restaurants* that store information about a specific city. A tourist may be interested in discovering the best combinations of hotels and restaurants in the same “Location”, by minimizing the hotel’s “Price”, maximizing the hotel’s “Rating”, maximizing the restaurant’s “Quality” and minimizing its “Distance” from a subway station. A sample SQL query that retrieves this information is the following<sup>1</sup>:

```
SELECT H.HID, R.RID, H.Hname, R.Rname, H.Location,
       H.Price, H.Rating, R.Distance, R.Quality
FROM Hotels H, Restaurants R
WHERE H.Location = R.Location
SKYLINE OF H.Price MIN, H.Rating MAX,
           R.Distance MIN, R.Quality MAX
```

The result of the query is depicted in Figure 1. As shown, the use of the skyline operator allows the retrieval of the most “important” join tuples according to the specified attributes.

The naive method of computing all join tuples first and then computing the skyline set leads to redundant processing, since the join is computed in its entirety (typically, an expensive operation) whereas the skyline set contains potentially only few tuples. Ideally, we would like to process and join only a subset of the input tuples in order to compute the skyline set of the join. This approach improves dramatically the performance of the naive method and makes the computation feasible in the context of large data sets. Unfortunately, it is non-trivial to decide which input tuples can be omitted without compromising the correctness of the computed skyline set. The reason is that the skyline set of the join depends both on the domination relationships in each input relation

<sup>1</sup>We assume that SQL is extended with a SKYLINE OF operator.

(e.g., which hotels have the best trade-off in rating and price, and which restaurants have the best trade-off in distance and quality), and on the join correlations across the two input relations (e.g., the existence of “good” hotels and restaurants in the same location). In general, the skyline set of the join is not necessarily identical to the join of the skyline sets of the input relations.

The majority of previous studies on skyline computation have focused on the case of a single relation [2, 4, 13]. The proposed techniques require the entire join result to be computed, and thus are prohibitively expensive as we explained earlier. Two recent studies have examined specifically the problem of computing the skyline over a join [7, 10]. However, we argue (and verify our claim experimentally) that these approaches are not effective. More specifically, the technique in [7] does not support early termination and hence it will always access all input tuples. The second technique, termed ProgXe [10], may terminate early under certain conditions, but it requires a specialized multi-dimensional access method whose parameters need to be tuned carefully. This requirement increases the overhead of integrating ProgXe in an existing system, and also precludes its usage in pipelined query plans where the inputs are generated by other physical operators. Moreover, ProgXe does not carry any theoretical guarantees on how early it can terminate, which may lead to unpredictable performance on certain inputs. This lack of robustness is particularly problematic in the context of ad-hoc data analysis, where good performance is critical.

**Our Contributions.** Motivated by the shortcomings of existing methods, we develop a novel algorithm, termed *SFSJ*, for computing the skyline set of a join  $R_1 \bowtie R_2$ . *SFSJ* is the first skyline join algorithm that combines several salient properties: (a) it accesses only a subset of each input relation and computes the skyline set without generating all the join results, (b) it can be readily implemented in existing database systems, as it relies on common infrastructure and does not require any complicated data structures, (c) it generates the skyline progressively, which facilitates its incorporation in pipelined query execution plans, and (d) it carries optimality and instance-optimality guarantees for its performance, thus ensuring that it works robustly across a variety of inputs. More concretely, the technical contributions of our work can be summarized as follows:

- We develop a novel early-termination condition (Section 4) that allows an algorithm to compute the correct skyline join set after reading only a subset of the input tuples. The condition is applied on a simple model of sorted input access, which is readily supported in existing systems using either B-trees over base tables or a sort operator over complex expressions.
- We introduce the *SFSJ* algorithm (Section 5) for computing the skyline over a join. *SFSJ* alternates between its inputs and generates the skyline tuples progressively as it computes the join results. It relies on the early-termination condition in order to determine when it has accessed enough tuples to generate the complete skyline set.
- We formally analyze the performance of *SFSJ* (Section 6) under different strategies that define the order in which the input relations are accessed. We introduce a new adaptive strategy that prioritizes access to the input relations and prove that it is optimal for *SFSJ*, i.e., it minimizes the total number of accessed tuples. Moreover, we provide a theoretical analysis that relies on techniques previously applied on the rank join problem [5, 11]. Specifically, we show that *SFSJ* is instance-optimal with a ratio of two within all deterministic algorithms and the class of inputs with bounded repetitions. To the best of our knowledge, ours is the first work to provide strong optimality results for the specific problem.

- We conduct a detailed experimental study to evaluate the performance of *SFSJ* (Section 7). Our results with synthetic and real-life data sets demonstrate that *SFSJ* computes the skyline join result efficiently after accessing only a subset of the input tuples. *SFSJ* outperforms two competitor techniques by a significant margin, thus validating its theoretical advantages over existing approaches.

## 2. RELATED WORK

**Single-Relation Skyline.** Previous studies have introduced a host of different techniques for computing the skyline of a single relation [1–4, 13]. Even though it is conceptually possible to apply these techniques on the results of a join operator, this solution is not likely to perform well in practice. Techniques that require an index over the input relation (e.g., [13]) will need the join results to be computed in full and indexed before computing the skyline. Techniques that do not require an index (e.g., [1, 2]) can be pipelined with the join operator, but they are decoupled from the semantics of the join and essentially examine the complete join stream before terminating. (We show this behavior analytically in Section 4.) In contrast, *SFSJ* fuses the computation of the join and the skyline in order to terminate early, i.e., *SFSJ* outputs the correct skyline without computing the full join or reading all the input tuples. Providing a condition for early termination in this setting is a key technical contribution of our work.

**Join Skyline.** Jin et al. [6] propose a multi-relational skyline operator that combines a sort-merge join algorithm with skyline processing. The operator involves a non-trivial processing cost, as it computes the skyline and group skyline tuples for all join values for each relation, then eliminates dominated tuples for each join value, and finally sorts the relations such that skyline tuples precede group skyline tuples. Clearly, the operator needs to access both relations multiple times to produce the correct result. Sun et al. [12] propose a distributed adaptation of SaLSa [1, 2] to compute the join skyline in a distributed setting. However, as will be shown in Section 4, this approach cannot lead to early termination. Moreover, the algorithms in [12] share the same restrictions with the approach proposed in [6]. In a follow-up study of [6], two non-blocking algorithms for skyline join are proposed [7]. The main idea is to interleave the join with the skyline computation, employing the nested-loop and sort-merge join algorithms. However, support for early termination is not provided. In contrast, our algorithm returns the correct result without accessing all input tuples.

In [10], the ProgXe framework is introduced for skyline join computation that supports progressive result generation. ProgXe partitions the input relations by using a multidimensional grid access method. This limits the framework to joins over base tables, whereas in practice the join may be also computed over complex relational expressions. Even in the case of base tables, the use of a multidimensional structure makes the performance of ProgXe sensitive to the number of dimensions, but most importantly to the resolution of the grid. ProgXe does not provide any method for optimizing its performance for two given relations (by setting the number of grid partitions to the most appropriate values), thus exhaustive testing is required. Further, as demonstrated in our experimental study, ProgXe does not terminate before exhausting both input relations in all common data distributions tested. Finally, the particular access method is not part of conventional database systems and hence requires an additional implementation effort. The techniques that we develop rely simply on sorted access, which is implementable with existing relational operators, and do not require tuning. Moreover, we are able to prove strong optimality properties about the early termination of our algorithms.

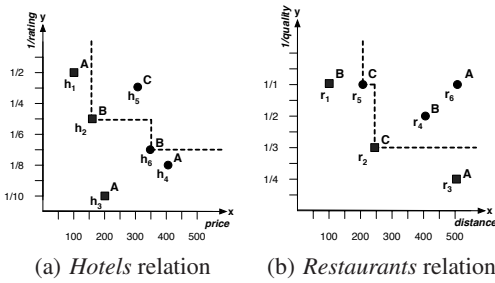


Figure 2: Relations *Hotels* and *Restaurants* from Figure 1.

Other works have also considered the issue of skyline join computation. The first work that implicitly deals with skyline join is from Koudas et al. [8]. Nevertheless, the proposed algorithms either do not support early termination or require multiple indexes over each input relation. In [9], a framework is proposed for preference evaluation inside the query processor, however their approach for skyline joins relies on [6]. Computing the skyline over a cartesian product of relations has been studied in [14], but the main focus is on identifying tuples of the individual relations that produce skyline join results, and not on the efficient computation of the skyline join.

### 3. PRELIMINARIES

Let  $R$  denote a relation and  $\mathcal{B}$  denote a set of numerical attributes in the schema of  $R$ . Let  $\tau$  and  $\tau'$  represent tuples in  $R$ . We say that tuple  $\tau'$  is *dominated* by  $\tau$  with respect to  $\mathcal{B}$ , denoted as  $\tau \prec_{\mathcal{B}} \tau'$ , if and only if  $\pi_{\mathcal{B}}(\tau') \neq \pi_{\mathcal{B}}(\tau)$  and  $\pi_{\beta}(\tau') \geq \pi_{\beta}(\tau)$  for all  $\beta \in \mathcal{B}$ . Accordingly, we define the *skyline set* of  $R$  with respect to  $\mathcal{B}$ , denoted  $SKY_{\mathcal{B}}(R)$ , as the minimal subset of  $R$  that dominates every other tuple in  $R$ , that is,  $\forall \tau' \in (R - SKY_{\mathcal{B}}(R)) \exists \tau \in SKY_{\mathcal{B}}(R)$  such that  $(\tau \prec_{\mathcal{B}} \tau')$ . We can extend our definitions based on maximizing values or a hybrid of the two models. We now introduce the *skyline join problem* that we address in this paper.

**DEFINITION 1. (Skyline Join Problem)** An instance  $I$  of the Skyline Join Problem is a tuple  $(R_1 \bowtie_{\text{eq}} R_2, \mathcal{B}_1, \mathcal{B}_2)$ , where  $R_1 \bowtie_{\text{eq}} R_2$  is an equi-join expression and  $\mathcal{B}_i$  is a set of numerical attributes from  $R_i$ ,  $i = 1, 2$ . The solution to an instance  $I$  is the set  $SKY_{\mathcal{B}_1 \cup \mathcal{B}_2}(R_1 \bowtie_{\text{eq}} R_2)$ .

As an example, the SQL query in Section 1 specifies an instance of the skyline join problem over relations *Hotels* and *Restaurants* and attribute-sets  $\mathcal{B}_1 = \{\text{Price, Rating}\}$  and  $\mathcal{B}_2 = \{\text{Distance, Quality}\}$  respectively.

The problem definition does not place any constraints on relations  $R_1$  and  $R_2$ . Indeed, they may result from complex relational expressions or simply be base tables. To facilitate exposition, we make two simplifying assumptions that are not crucial for our techniques. First, we assume that the join expression is a natural join on a single common attribute  $\alpha$ , and henceforth write  $R_1 \bowtie R_2$ . We use  $\tau_i.\alpha$  to denote the value of the join attribute for a tuple  $\tau_i \in R_i$ ,  $i = 1, 2$ . Second, we assume that the domain of any attribute in  $\mathcal{B}_1 \cup \mathcal{B}_2$  is the interval  $[0, 1]$ , and use the term skyline or scoring attribute to refer to any  $\beta \in \mathcal{B}_i$ . We also simplify our notation by omitting the attribute-set from the skyline notation. Hence, for an instance  $I = (R_1, R_2, \mathcal{B}_1, \mathcal{B}_2)$ , the skyline set  $SKY(R_1 \bowtie R_2)$  is taken with respect to  $\mathcal{B}_1 \cup \mathcal{B}_2$ . Similarly, the skyline set  $SKY(R_i)$  is taken with respect to  $\mathcal{B}_i$ . We extend this simplification to the notation for tuple dominance.

On a final note, we introduce the auxiliary concept of *group skyline* which is used in the development of our techniques.

**DEFINITION 2. (Group Skyline)** Given an instance  $I$  of the skyline join problem, the group skyline of  $R_i$  with respect to value  $x$  of the joining attribute is defined as  $SKY(R_i, x) = SKY(\sigma_{\alpha=x}(R_i))$ .

A consequence of this definition is that if a tuple  $\tau_i \in R_i$  appears in  $SKY(R_i)$  then it also appears in  $SKY(R_i, x)$  for  $x = \tau_i.\alpha$ . For example, Figure 2(a) depicts the tuples of relation *Hotels* (see Figure 1) with respect to the set  $\mathcal{B} = \{\text{Price, Rating}\}$ . Each tuple is mapped to a point in the 2-dimensional space defined by the value of the price and the rating of the hotel. The point is annotated with the tuple id and the corresponding value of the join attribute. Skyline tuples are depicted with squares. We also use a dashed line to connect the group skyline tuples for join value  $B$ . Notice that the skyline tuple  $h_2$  is also a group skyline tuple for  $B$ . Figure 2(b) shows a similar visualization for relation *Restaurants*. Group skylines enable the formulation of a very useful property.

**PROPERTY 1.** Let  $\tau \equiv \tau_1 \bowtie \tau_2$  be a tuple in  $R_1 \bowtie R_2$ . It holds that  $\tau \notin SKY(R_1 \bowtie R_2)$  if  $\exists i : \tau_i \notin SKY(R_i, \tau_i.\alpha)$

Essentially, tuples  $\tau_i \in R_i$  that are not group skyline tuples cannot contribute to the join skyline. The reason is that the produced join tuples are guaranteed to be dominated. Returning to the example of Figure 2(a),  $h_4$  cannot produce a skyline join result, since  $h_3 \prec h_4$  and  $h_3 \in SKY(\text{Hotels}, A)$ , and hence any join result  $h_4 \bowtie r$  (e.g., for  $r = r_6$ ) will be dominated by the result  $h_3 \bowtie r$ . Furthermore, the joined tuple that consists of a skyline tuple (e.g.,  $r_1$ ) joined with a group skyline tuple (e.g.,  $h_6$ ) is always a skyline join result. Notice that we cannot ascertain whether the combination of group skyline tuples produces skyline join results beforehand.

### 4. EARLY TERMINATION

Given an instance  $I$  of the skyline join problem, it is always possible to compute the set  $SKY(R_1 \bowtie R_2)$  by first generating all tuples of  $R_1 \bowtie R_2$  and then applying a conventional single-relation skyline algorithm [3, 4]. Obviously, this method leads to redundant processing, and hence bad performance, since it requires accessing all tuples of both relations and generate all join results, in order to return the potentially few skyline tuples.

In contrast, the high-level idea of our approach is to compute the  $SKY(R_1 \bowtie R_2)$  by *accessing only a prefix of  $R_1$  and  $R_2$* , assuming that their tuples are accessed in a sorted order defined by the skyline attributes. We refer to this feature of a skyline join algorithm as *early termination* to differentiate from algorithms that need to access all tuples of both relations  $R_i$  to compute the skyline join result. Early termination is desirable because it results in savings in I/O (fewer tuples are accessed) and computation (fewer join results are generated and fewer dominance tests are required), and can thus improve substantially on the performance of techniques that exhaust their inputs.

More specifically, we assume that each relation  $R_i$  is accessed one tuple at a time, in an ascending order according to the following function:

$$fmin(\tau_i) = \min_{\beta \in \mathcal{B}_i} \{\pi_{\beta}(\tau_i)\}, \quad (1)$$

for  $i = 1, 2$ . Figure 3 shows an illustration of this sorted access model. Essentially, the tuples are organized in “bands” of increasing  $fmin$  value. Hence, an algorithm will need to exhaust all the tuples in the current band before observing an increase in the  $fmin$  value. Moreover, the corner point of each band dominates all the tuples in higher bands. This observation plays a central role in the development of our techniques.

A key advantage of this access model is that it can be implemented easily in an existing system, using common infrastructure

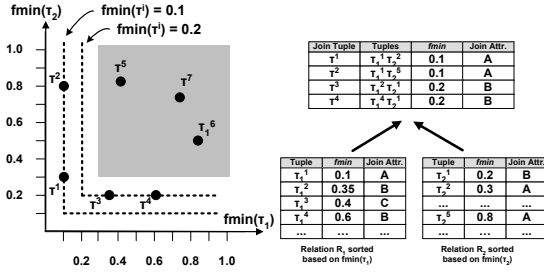


Figure 3: Example of SaLSa on relation  $R_1 \bowtie R_2$ .

such as secondary B-tree indexes. We discuss the details at the end of the section, but for the time being we note that the overhead is much smaller compared to the specialized access methods required by other techniques [10].

In this section, we formalize a condition that allows an algorithm to check whether  $SKY(R_1 \bowtie R_2) = SKY(S_1 \bowtie S_2)$ , where  $S_i$  denotes a prefix of  $R_i$  according to the sorted order. Clearly, this is the crucial prerequisite in order to support early termination. The condition that we introduce is novel in that it takes into account the semantics of both the join and the skyline operator, and thus differs from previous works that consider solely the skyline operator.

#### 4.1 Inadequacy of Existing Techniques

Before presenting the details of the new early-termination condition, we examine if the algorithms that employ sorted access to solve the single-relation skyline problem [1, 2, 13] can be adapted to the skyline join problem. Our analysis yields a negative answer: existing algorithms employ a stopping condition that is oblivious to the semantics of the join operator, and hence fails to effectively support early termination.

The analysis that we present is based on the state-of-the-art SaLSa algorithm of Bartolini et al. [1, 2]. SaLSa receives as input a single stream of tuples that are sorted based on  $fmin$ , and outputs the skyline set of the input tuples. The algorithm accesses its input one-tuple-at-a-time, and maintains the current skyline set and an associated threshold  $x$ . The current skyline set is returned as the output once the next input tuple has an  $fmin$  value that is not smaller than  $x$ .

In the context of the skyline join problem, we must couple SaLSa with a join algorithm, say  $A$ , that reads  $R_1$  and  $R_2$  in sorted order of  $fmin$  and progressively computes  $R_1 \bowtie R_2$  also in sorted order of  $fmin$ . For instance,  $A$  can be a rank join algorithm that employs  $fmin$  as the scoring function. Algorithm  $A$  returns the next join tuple as requested by SaLSa, and terminates when SaLSa satisfies the aforementioned termination condition.

Unfortunately, the previous scheme is not effective. We prove that any correct algorithm  $A$  will have to exhaust at least  $R_1$  or  $R_2$  if  $SKY(R_1 \bowtie R_2)$  is a non-trivial skyline set. Formally speaking, a trivial skyline set satisfies the following property:  $\tau.\beta = \rho.\beta$  for all  $\beta \in \mathcal{B}_1 \cup \mathcal{B}_2$  and for all  $\tau$  and  $\rho$  in  $SKY(R_1 \bowtie R_2)$ . Essentially, all the skyline join tuples are identical and correspond to a single point that dominates every other join tuple. In practice we expect to see non-trivial skylines, in which case  $A$  will have to exhaust at least one input.

**THEOREM 1.** *Let  $A$  be a correct deterministic algorithm that computes  $R_1 \bowtie R_2$  in ascending order of  $fmin$  (using some arbitrary tie breaking rule), assuming that  $R_1$  and  $R_2$  are also accessed in sorted order of  $fmin$ . Suppose that SaLSa reads the output of  $A$  and let  $\tau^{\text{last}}$  be the last tuple that SaLSa reads before it returns the skyline  $SKY(R_1 \bowtie R_2)$ . If  $SKY(R_1 \bowtie R_2)$  is non-trivial, then*

*$A$  has to exhaust at least one of its inputs in order to generate the results up to  $\tau^{\text{last}}$ .*

**PROOF.** Let  $\tau^{\text{first}} \equiv \tau_1 \bowtie \tau_2$  be the first tuple accessed by SaLSa. Clearly,  $fmin(\tau^{\text{first}}) \leq fmin(\tau^{\text{last}})$ . Without loss of generality, assume that  $fmin(\tau) = fmin(\tau_1)$ . The proof is symmetric if  $fmin(\tau) = fmin(\tau_2)$ .

We prove first an auxiliary result, namely that  $fmin(\tau^{\text{first}}) < fmin(\tau^{\text{last}})$ . Suppose that this is false, i.e.,  $fmin(\tau^{\text{first}}) = fmin(\tau^{\text{last}})$ . Since SaLSa terminates after reading  $\tau^{\text{last}}$ , it holds that  $fmin(\tau^{\text{last}}) \geq x$ , where  $x$  is the threshold maintained by SaLSa. By definition,  $x = \max\{\rho.\beta | \beta \in \mathcal{B}_1 \cup \mathcal{B}_2\}$ , where  $\rho$  is the tuple in the current skyline set that has the minimum maximum coordinate, i.e.,  $\rho = \arg \min\{\max\{\tau.\beta | \beta \in \mathcal{B}_1 \cup \mathcal{B}_2\} | \tau \text{ is a skyline tuple}\}$ . From our hypothesis, it holds that  $x = \max\{\rho.\beta | \beta \in \mathcal{B}_1 \cup \mathcal{B}_2\} \leq fmin(\tau^{\text{first}})$ , and since  $x \geq fmin(\rho) \geq fmin(\tau^{\text{first}})$ , it follows that  $x = \max\{\rho.\beta | \beta \in \mathcal{B}_1 \cup \mathcal{B}_2\} = fmin(\tau^{\text{first}}) = fmin(\rho)$ . This means that  $\rho$  corresponds to the corner-point  $(fmin(\tau^{\text{first}}), \dots, fmin(\tau^{\text{first}}))$  and hence it dominates every other tuple in  $R_1 \bowtie R_2$ . This is a contradiction, as we assumed that  $SKY(R_1 \bowtie R_2)$  is non-trivial. Hence, it must be that  $fmin(\tau^{\text{first}}) < fmin(\tau^{\text{last}})$ .

We proceed to show that  $A$  will exhaust  $R_2$ . Suppose otherwise. Since  $fmin(\tau^{\text{last}}) > fmin(\tau^{\text{first}})$ , this means that  $A$  has generated all the join results whose  $fmin$  value is equal to  $fmin(\tau^{\text{first}})$  without having accessed the last tuple of  $R_2$ . It is straightforward to show that  $A$  commits a mistake if that last tuple joins with  $\tau_1$ , which contradicts our assumption that  $A$  is a correct algorithm.  $\square$

Figure 3 shows an example of applying SaLSa for computing the skyline join set<sup>2</sup>. The x-axis and y-axis represent the  $fmin(\tau_i)$  values with respect to  $R_1$  and  $R_2$  respectively.  $R_i$  is sorted based on  $fmin(\tau_i)$ ,  $i = 1, 2$ , but the join tuples are accessed by SaLSa with respect to  $fmin(\tau)$ . Henceforth, we use the superscript of a tuple to denote its rank in the corresponding sorted relation. SaLSa would first process tuple  $\tau^1$  with  $fmin(\tau^1) = 0.1$  and set a threshold  $x = \max_{\beta \in \mathcal{B}_1 \cup \mathcal{B}_2} \{\pi_\beta(\tau^1)\} = 0.3$ . Based on the termination condition of SaLSa, tuples  $\tau^i$  are accessed until  $fmin(\tau^i) \geq x$ . Thus, in Figure 3, all tuples enclosed in the shadowed rectangle are pruned. On the other hand, all join tuples that fall on a dashed line have the same  $fmin$  value. For example, tuples  $\tau^1$  and  $\tau^2$  have the same  $fmin$  value, and they are produced by the first tuple  $\tau_1^1$  of  $R_1$  and tuples  $\tau_2^2$  and  $\tau_2^5$  of  $R_2$ . In general, the first tuple of  $R_1$  can be combined with the last tuple of  $R_2$ , as long as they have the same join value and the generated join tuple will have the minimum  $fmin$  score. Thus, SaLSa needs to access all tuples of  $R_2$  in order to ensure that no other tuple  $\tau_2^i$  can be joined with  $\tau_1^1$ .

The behavior of SaLSa is clearly problematic. Performance is impacted negatively, particularly if the exhausted input corresponds to a large relation. Moreover, the trigger for this bad performance is a high-level property (non-trivial skyline sets) that is quite likely to appear in practice. The proof reveals that the problem lies with the design of SaLSa's termination condition, which is tailored to the single-relation problem and is thus oblivious of the underlying join operation. In contrast, we develop an early termination condition that directly takes into account the semantics of the join operation, and hence avoids the pitfalls of SaLSa's termination condition.

#### 4.2 Condition for Early Termination

We develop a condition that asserts the following:  $SKY(R_1 \bowtie R_2)$  is equal to  $SKY(S_1 \bowtie S_2)$ , where each  $S_i$  denotes a prefix of  $R_i$  according to the sorted access model. We first provide a

<sup>2</sup>Recall that the subscript of a tuple denotes the relation the tuple belongs to, whereas the absence of subscript denotes a join tuple.

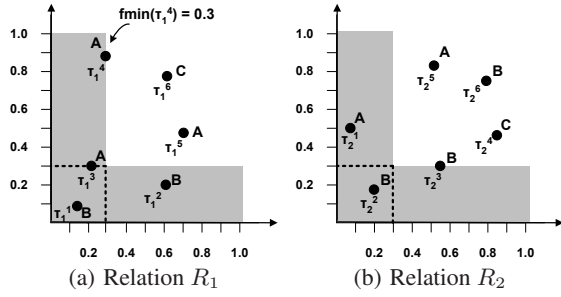


Figure 4: Example relations  $R_1$  and  $R_2$ .

precise definition of the prefix that is considered in the termination condition.

**DEFINITION 3. (Closed Prefix of Relation  $R_i$ )** Let  $S_i$  be a non-empty prefix of relation  $R_i$ , and define  $t(S_i) = \max\{fmin(\rho_i) \mid \rho_i \in S_i\}$ .  $S_i$  is called a closed prefix of  $R_i$  iff  $fmin(\tau_i) > t(S_i)$  for all  $\tau_i \in R_i - S_i$ .

Essentially, the tuples in  $R_i - S_i$  correspond to bands with strictly higher  $fmin$  values than all tuples in the closed prefix  $S_i$ . An example is shown in Figure 4 for two sample relations  $R_1$  and  $R_2$ . We assume that each relation has two skyline attributes. We can assert that  $S_1 = \{\tau_1^1, \tau_1^2, \tau_1^3, \tau_1^4\}$  forms a closed prefix for  $R_1$  according to the previous definition, since every other tuple in  $R_1$  has a  $fmin$  value that is higher than  $t(S_1) = \max\{fmin(\tau_1^1), fmin(\tau_1^2), fmin(\tau_1^3), fmin(\tau_1^4)\}$ . A similar observation can be made for  $S_2 = \{\tau_2^1, \tau_2^2, \tau_2^3\}$ .

The significance of the closed prefix  $S_i$  is that it defines a point, namely the corner of the last accessed band, that dominates all tuples in  $R_i - S_i$ . Indeed, the point  $(t(S_i), t(S_i), \dots, t(S_i))$  dominates all tuples in  $R_i - S_i$ . In turn, any tuple that dominates this corner also dominates  $R_i - S_i$ . We capture this property through the *minimum anti-dominance region*  $MAD(S_i)$ .

**DEFINITION 4. (Minimum Anti-Dominance Region)** The minimum anti-dominance region of a closed prefix  $S_i$  is defined as  $MAD(S_i) = \{\rho_i \mid \rho_i \in S_i \wedge \rho_i.\beta \leq t(S_i) \forall \beta \in B_i\}$ .

The dashed rectangles in Figure 4 depict  $MAD(S_i)$ , assuming that  $S_i$  contains the tuples in the corresponding shaded region.

Assuming that  $MAD(S_i) \neq \emptyset$ , none of the tuples in  $R_i - S_i$  can appear in  $SKY(R_i)$ , and hence we can assert that  $SKY(R_i) = SKY(S_i)$ . This can be verified on the example of Figure 4. Nevertheless, this condition is not sufficient for stopping accessing tuples of  $R_i$ , because group skyline tuples (which may not appear in  $SKY(R_i)$ ) may also contribute to  $SKY(R_1 \bowtie R_2)$  (see the properties of the skyline join discussed in Section 3). A correct alternative would be to stop accessing  $R_i$  only if  $S_i$  contains all the group skyline tuples. However, this approach is not viable in practice, because the number of distinct join values may not be known. In this case, all the tuples in  $R_i$  have to be accessed in order to ensure that no other group skyline tuple exists. Moreover, the cost of computing all group skyline tuples can be prohibitively high (if the domain of join values is of high cardinality), while some group skyline tuples may not contribute to the skyline join result at all (for example, this is case for join value  $C$  in Figure 4). Therefore, the main challenge is to find an early-termination condition that does not require computing the group skyline sets for all join values. We achieve this goal by taking advantage of the properties of  $MAD(S_i)$ , that lead to the formulation of a sufficient condition for when a tuple in  $R_i - S_i$  can generate a skyline join result.

**LEMMA 1.** If  $MAD(S_j) \neq \emptyset$  and  $\pi_\alpha(SKY(S_j)) - \pi_\alpha(MAD(S_i)) = \emptyset$  for  $i \neq j$  then there exists no tuple in  $R_i - S_i$  that can generate a skyline join result, i.e.,  $SKY(R_i \bowtie R_j) \cap ((R_i - S_i) \bowtie R_j) = \emptyset$ .

**PROOF.** By contradiction. Suppose that there exists a  $\tau \equiv \tau_i \bowtie \tau_j$  that belongs to  $SKY(R_i \bowtie R_j) \cap ((R_i - S_i) \bowtie R_j)$ .

The fact that  $MAD(S_j) \neq \emptyset$  implies that  $SKY(S_j) = SKY(R_j)$ . It follows that  $MAD(S_i) \neq \emptyset$ , because  $SKY(S_j) \neq \emptyset$  and otherwise the condition would not be true. We consider two cases for  $\tau_i$ .

**Case 1:**  $\tau_i.\alpha \in \pi_\alpha(MAD(S_i))$ . It follows that there exists  $\rho_i \in MAD(S_i)$  such that  $\tau_i.\alpha = \rho_i.\alpha$ . Clearly,  $\rho_i \bowtie \tau_j$  is a valid join result, and it also holds that  $\rho_i \prec \tau_i$  due to the dominance property of  $MAD(S_i)$ . Hence,  $\rho_i \bowtie \tau_j \prec \tau_i \bowtie \tau_j$ .

**Case 2:**  $\tau_i.\alpha \notin \pi_\alpha(MAD(S_i))$ . Since  $\tau_j.\alpha = \tau_i.\alpha$  and  $\pi_\alpha(SKY(S_j)) \subseteq \pi_\alpha(MAD(S_i))$ , it follows that  $\tau_j.\alpha \notin \pi_\alpha(SKY(S_j))$  and hence  $\tau_j \notin SKY(S_j)$ . Given that  $SKY(S_j) = SKY(R_j)$ , there exists  $\tau'_j \in SKY(S_j)$  such that  $\tau'_j \prec \tau_j$ . The condition ensures that  $MAD(S_i)$  contains a tuple  $\tau'_i$  such that  $\tau'_i.\alpha = \tau'_j.\alpha$ . Since  $\tau'_i \prec \tau_i$  (by the dominance property of  $MAD(S_i)$ ), it holds that  $\tau'_i \bowtie \tau'_j \prec \tau_i \bowtie \tau_j$ .

In both cases, we identify a join tuple that dominates  $\tau_i \bowtie \tau_j$ , which contradicts our assumption that  $\tau_i \bowtie \tau_j \in SKY(R_i \bowtie R_j)$ .  $\square$

Lemma 1 states that if for every skyline tuple in  $SKY(S_1)$  ( $j = 1$ ) there exists a tuple in  $MAD(S_2)$  ( $i = 2$ ) with same join value, then the tuples in  $R_2 - S_2$  do not contribute to the skyline join result. We illustrate the basic intuition with a simple example.

Assume two group skyline tuples  $\rho_1 \in SKY(R_1, \beta)$ ,  $\rho_2 \in SKY(R_2, \beta)$  of  $R_1$  and  $R_2$  respectively, such that  $\rho_2 \in R_2 - S_2$ . Let us first assume that  $\rho_1$  is also a skyline tuple ( $\rho_1 \in SKY(R_1)$ ). Since  $MAD(S_1) \neq \emptyset$ , it holds that  $\rho_1 \in SKY(S_1)$  and that there exists a tuple  $\tau_2 \in MAD(S_2)$  such that  $\rho_1.\alpha = \tau_2.\alpha = \beta$ . This leads to a contradiction, since  $\tau_2$  dominates  $\rho_2$  and therefore  $\rho_2$  is not a group skyline tuple and does not contribute to the skyline join result. On the other hand, if  $\rho_1 \notin SKY(R_1)$ , then there exists a tuple  $\tau_1 \in SKY(S_1) = SKY(R_1)$  such that  $\tau_1$  dominates  $\rho_1$ . Based on the condition of Lemma 1, there exists also a tuple  $\tau_2 \in MAD(S_2)$  such that  $\tau_1.\alpha = \tau_2.\alpha = \beta$ . Furthermore,  $\tau_2$  dominates  $\rho_2$ , since it dominates all tuples in  $R_2 - S_2$ . We can thus verify that  $\tau_1 \bowtie \tau_2 \prec \rho_1 \bowtie \rho_2$ , since  $\tau_1 \prec \rho_1$  and  $\tau_2 \prec \rho_2$ . Hence, any tuple in  $R_2 - S_2$  is not relevant for the computation of  $SKY(R_1 \bowtie R_2)$ , and this is independent of whether  $R_2 - S_2$  contains additional group skyline tuples  $SKY(R_2, \beta)$ .

Overall, the join values that appear in  $MAD(S_i)$  and  $SKY(S_i)$  ( $i = 1, 2$ ) are sufficient for deciding whether it holds that  $SKY(S_1 \bowtie S_2) = SKY(R_1 \bowtie R_2)$ . Building on Lemma 1, it is straightforward to define our early-termination condition.

**THEOREM 2. (Early Termination Condition)** It holds that  $SKY(R_1 \bowtie R_2) = SKY(S_1 \bowtie S_2)$  if the following holds:

$$\pi_\alpha(SKY(S_1)) - \pi_\alpha(MAD(S_2)) = \emptyset \wedge$$

$$\pi_\alpha(SKY(S_2)) - \pi_\alpha(MAD(S_1)) = \emptyset$$

**PROOF.** Follows directly by applying Lemma 1 for each conjunct.  $\square$

We further illustrate Theorem 2 using the example of Figure 4. We first verify that the condition for early termination holds in this example, and then we explain why no combination of any group skyline tuple can produce additional skyline join results. The early termination condition demands that for every join value of any tuple

in  $SKY(S_i)$ , there exists a tuple in  $MAD(S_j)$  with the same join value. Indeed, looking at Figure 4, for  $\tau_1^1 \in SKY(R_1)$  there exists the tuple  $\tau_2^1 \in MAD(S_2)$ , and for  $\tau_2^2, \tau_2^3 \in SKY(R_2)$  the tuples  $\tau_1^3, \tau_1^1 \in MAD(S_1)$  respectively. Based on the early termination condition, no skyline results can be generated by tuples in  $R_1 - S_1$  and  $R_2 - S_2$ .

Now, let us consider group skyline tuples. Since  $MAD(S_i) \neq \emptyset$  (for  $i = 1, 2$ ), all join results produced by the skyline tuples of  $R_1$  and  $R_2$  have been generated. Skyline tuples of  $R_i$  can be also combined with group skyline tuples of  $R_j$  to produce skyline join results. Given a tuple  $\tau_i \in SKY(R_i)$  with join value  $\beta$ , there exists a tuple in  $MAD(S_j)$  with the same join value. Therefore, no group skyline tuple  $SKY(R_j, \beta)$  can be found in  $R_j - S_j$ , if there exists a tuple  $\tau_i \in SKY(R_i)$  with join value  $\beta$ . For example, in Figure 4, the tuple  $\tau_1^1 \in SKY(R_1)$  has the join value  $B$ , which is the join value of tuple  $\tau_2^2 \in MAD(S_2)$ . Then, no group skyline tuple for  $B$  can be found in  $R_2 - S_2$ , since any such tuple is dominated by  $\tau_2^2$ . Finally, group skyline tuples of  $R_1, R_2$  that have a join value different than those that appear in the tuples of  $SKY(R_1)$  and  $SKY(R_2)$  may produce skyline join results. A group skyline tuple of  $R_1$ , in our example  $\tau_1^6$ , has a dominating tuple  $\tau_1^1 \in SKY(R_1)$ . Clearly, the join result  $\tau_1^6 \bowtie \tau_2^4$  is dominated by  $\tau_1^1 \bowtie \tau_2^2$  and hence cannot appear in  $SKY(R_1 \bowtie R_2)$ . Thus, our early termination condition does not require to retrieve all group skyline tuples for all join values.

### 4.3 Implementing Sorted Access

This subsection examines the implementation of the sorted access model. First, we consider the case where  $R_i$  is a base table. One obvious solution is to maintain a B-tree index on  $R_i$  for each possible choice of  $B_i$ , using the result of  $fmin$  as the key of the index. This approach is feasible if there are relatively few choices for  $B_i$  and the relation is not updated frequently.

An alternative approach is to realize the sorted access through several B-tree indices. More specifically, let  $\mathcal{B}$  be the set of attributes of  $R_i$  that can be potentially used as scoring attributes. For each  $\beta \in \mathcal{B}$ , we maintain a separate index that uses  $\beta$  as its key and the remaining attributes in  $\mathcal{B}$  as “follow” attributes. Hence, a tuple  $\tau_i$  is mapped to an index entry that records all the attributes of  $\mathcal{B}$  and the corresponding record identifier (RID), and the index entries are sorted based on  $\beta$ . This means that the index can provide a stream of RIDs sorted on  $\beta$ . Given a specific choice for  $B_i \subseteq \mathcal{B}$ , we open cursors to the corresponding indices and essentially perform a merge of the RID streams in a single RID stream, which can then be transformed to a sorted tuple stream. To avoid the generation of duplicates, an RID is returned from an index scan only if the key value is the minimum of all attributes in  $B_i$  for the corresponding tuple. This check can be performed efficiently, since all attributes in  $B_i$  can be found in the follow attributes of the index entry. Even though this scheme is not as efficient as accessing  $R_i$  through a single index, the advantage is that the number of indices is small, equal to the number of attributes in  $\mathcal{B}$ .

Next, let us examine the case where  $R_i$  is generated by a relational expression. One obvious solution is to sort  $R_i$  based on  $B_i$ , but this would require scanning all of  $R_i$ . Another choice is to evaluate the expression using a physical plan that generates  $R_i$  in the correct order. For instance, assume that  $R_i$  is the result of applying a selection predicate on a base table  $T$ . If  $T$  is accessed based on one of the previous schemes, then  $R_i$  can be generated by applying the predicate on-the-fly on the tuples accessed from  $T$ .

## 5. THE SFSJ ALGORITHM

In this section, we introduce a novel skyline join algorithm termed

### Function SFSJ( $I$ )

**Input:** An instance  $I = (R_1, R_2, B_1, B_2)$  of the skyline join problem such that  $I \in \mathcal{I}$ .

**Output:** The skyline  $SKY(R_1 \bowtie R_2)$ .

**Data:** Tuple-sets  $S_1$  and  $S_2$  for  $R_1$  and  $R_2$  respectively initialized to  $\emptyset$ ; Current groups  $B_1$  and  $B_2$ ; Tuple-sets  $M_1$  and  $M_2$  initialized to  $\emptyset$ ; Current bounds  $t_1$  and  $t_2$  initialized to  $-\infty$ .

**Parameters:** Deterministic pulling strategy  $P$ .

```

1 halt ← false;
2 while !halt do
3   i ← next input index from P ;
4   j ← {1, 2} − {i} ;
5   τi ← next tuple from Ri ;
6   if fmin(τi) = ti then
7     Bi ← Bi ∪ {τi} ;
8   else
9     for ρi ∈ Bi do
10      if ∄ρ'i ∈ Si ∪ Bi: ρ'i ≺ ρi and ρ'i.α = ρi.α then
11        add ρi to Si ;
12        H ← H ∪ {ρi} ⋈ Sj ;
13      for (τ ∈ H) do
14        if ∄τ' ∈ H ∪ O : τ' ≺ τ then add τ to O;
15      for τ'i ∈ Si: πβ(τ'i) ≤ fmin(τi) for all β ∈ Bi do
16        add τ'i to Mi;
17      halt ← Evaluate condition of Theorem 2 using Mi for
18        MAD(Si);
19      Bi ← {τi}, ti ← fmin(τi);
19 return O;
```

Figure 5: The SFSJ algorithm.

*SFSJ* (short for *Sort-First-Skyline-Join*). At a high-level, *SFSJ* alternates between relations  $R_1$  and  $R_2$  and employs Theorem 2 to check whether the accessed tuples are sufficient to compute the correct skyline join result. An important feature is that it generates skyline tuples as soon as possible, even before the early termination condition is satisfied. Moreover, it prunes input tuples if they cannot contribute to the join skyline, thus reducing the number of generated join results and dominance tests.

**Algorithm Description.** Figure 5 presents the pseudocode for *SFSJ*. We first describe the data structures used by the algorithm. For each input  $R_i$ , *SFSJ* maintains a tuple-set  $S_i$  that is conceptually equivalent to a closed prefix of the input. Accordingly, a tuple-set  $M_i$  is used to record the current anti-dominance region based on  $S_i$ . Finally, the algorithm maintains a tuple-set  $B_i$  that maintains all the recently accessed tuples with the same  $fmin$  value, which is recorded in variable  $t_i$ . This tuple-set enables *SFSJ* to process for each relation  $R_i$  groups of tuples with the same  $fmin$  value.

Moreover, the algorithm is parameterized by a deterministic pulling strategy  $P$  that determines the order in which *SFSJ* accesses its inputs. One obvious choice is to employ simple round-robin access, but it is also possible to favor one input over the other (e.g., pull from  $R_1$  twice as much as from  $R_2$ ), or determine the next input to pull adaptively based on the observed data values. We analyze possible pulling strategies in the next section. We do not make any further assumptions about  $P$ , except that it always returns an index  $i$  such that  $R_i$  is not yet exhausted.

*SFSJ* works in iterations, where in each iteration it accesses a new tuple from a relation  $R_i$  that is determined by the strategy  $P$  (line 3). If the newly accessed tuple  $\tau_i$  satisfies  $fmin(\tau_i) = t_i$  (line 6), then  $\tau_i$  is added to the current batch  $B_i$  and the algorithm continues with the next iteration. Otherwise,  $\tau_i$  signals the start of a new group, and the tuples in  $B_i$  are integrated in  $S_i$  and new join tuples are produced (lines 9–12). The batch-update of  $S_i$  ensures that  $S_i$  is always a valid closed prefix of  $R_i$ , as defined by Definition 3. Furthermore, as an optimization, *SFSJ* adds to  $S_i$  only group sky-

line tuples (line 10), since otherwise they cannot contribute to the skyline join result (Property 1). Subsequently, each tuple  $\rho_i$  that is added to  $S_i$  is joined with the tuples in  $S_j$  ( $j \neq i$ ), and any join results are kept in set  $H$  (line 12).

After all tuples in  $B_i$  have been processed, the generated tuples in  $H$  are tested for dominance against the tuples in  $H \cup O$ , and the non-dominated tuples are added to  $O$  (lines 13–14). *SFSJ* can report the skyline join tuples progressively, as soon as they are added to  $O$  (line 14). Any tuple in  $R_i - S_i$  has an *fmin* value greater than  $t_i$ , and therefore (as we will prove in Lemma 2) cannot produce join tuples that dominate any of the existing tuples in  $O$ . The next and final step is to update the anti-dominance region  $M_i$  based on the updated  $S_i$  (lines 15–16). Once  $M_i$  has been updated, the algorithm checks the stopping condition (line 17) to determine whether it is safe to terminate early.

**Correctness.** As already mentioned, *SFSJ* relies on the condition of Theorem 2 in order to terminate early with a correct solution. However, it deviates from a naive application of the condition in two important ways: it reports skyline join results progressively, and it prunes tuples that are not group skyline tuples. We show that these features do not affect correctness.

First, we prove a key lemma for progressive result generation.

**LEMMA 2.** *Let  $\tau = \tau_1 \bowtie \tau_2$  and  $\tau' = \tau'_1 \bowtie \tau'_2$  that appear in  $R_1 \bowtie R_2$ . It holds that  $\tau'_i \not\prec \tau_i$  if  $\exists i$  such that  $fmin(\tau_i) < fmin(\tau'_i)$ .*

**PROOF.** It holds that  $fmin(\tau_i) < fmin(\tau'_i)$  and therefore there exists at least one skyline attribute  $\beta$  for which  $\tau_i.\beta < \tau'_i.\beta$ . It follows that  $\tau'$  cannot dominate  $\tau$ .  $\square$

The above lemma guarantees that if every relation  $R_i$  is accessed in ascending order of the *fmin* values and all tuples with the same *fmin* value are processed in a batch, then the join results of a tuple  $\tau'_i$  cannot dominate those of tuple  $\tau_i$ , if  $\tau_i$  is in a batch that precedes the batch of  $\tau'_i$ . This enables the progressiveness of our algorithm.

Our second result states the overall correctness of *SFSJ*.

**THEOREM 3.** *When the stopping condition holds, *SFSJ* has obtained the correct skyline join result.*

**PROOF.** Theorem 2 proves that the stopping condition always guarantees that the complete and correct skyline join result set is computed, if  $S_1$  and  $S_2$  are closed prefixes of  $R_1$  and  $R_2$  respectively. Lemma 2 proves that only skyline join tuples are added to the result set (lines 13–14). Finally, the correctness of *SFSJ* is not violated by discarding tuples that are not group skyline tuples (line 10). Property 1 guarantees that all tuples of  $S_1$  and  $S_2$  that may produce skyline join tuples have been processed, since only tuples that do not belong to any group skyline set are discarded. Furthermore, the stopping condition cannot be satisfied earlier due to discarding of tuples of  $S_i$ , since (a) the skyline set  $SKY(S_i)$  does not change by discarding dominated tuples, and (b) the distinct values of the joining attribute in  $M_i$  are not influenced by discarding tuples that are not group skyline tuples (a tuple  $\tau_i$  is added to  $M_i$ , unless there exists a tuple  $\tau'_i \in M_i$  such that  $\tau_i.\alpha = \tau'_i.\alpha$  and  $\tau'_i \prec \tau_i$ ). Therefore, we conclude that *SFSJ* can safely stop processing tuples when the stopping condition holds, even though only group skyline tuples are maintained in the sets  $S_i$ .  $\square$

**Implementation Details.** We discuss next some important details about the efficient maintenance of the tuple-sets used by *SFSJ*, namely  $S_i$ ,  $M_i$ , and  $SKY(S_i)$ . The key observation is that we can efficiently maintain these tuple-sets by taking advantage of the

property that they are append-only. This property stems from the fact that each relation  $R_i$  is accessed in ascending order of *fmin*.

In more detail, the set of tuples  $S_i$  of relation  $R_i$  is implemented as a sorted list of tuples sorted by  $fmax(\tau_i) = \max_{\beta \in \mathcal{B}_i} \{\pi_\beta(\tau_i)\}$ . For any two tuples  $\tau_i, \tau'_i \in S_i$ , if  $fmax(\tau_i) < fmax(\tau'_i)$ , then  $\tau_i$  precedes  $\tau'_i$  in the list  $S_i$ . This technique allows efficient implementation of  $M_i$ , namely as a pointer to a tuple  $\hat{\tau}_i \in S_i$ . Given the last processed tuple  $\tau_i$ ,  $M_i$  is updated by examining the tuples starting from  $\hat{\tau}_i$  and stopping at the first tuple  $\tau'_i$  for which  $fmax(\tau'_i) < fmin(\tau_i)$ . The tuple  $\hat{\tau}_i$  preceding  $\tau'_i$  is the new position for the pointer that delineates  $M_i$ . Then,  $M_i$  is a subset of the tuple-set  $S_i$  containing all tuples starting from the first tuple of  $S_i$  up to the tuple  $\hat{\tau}_i$ . For example, consider the relation  $R_2$  depicted in Figure 4(b), where  $\tau_2^3$  is the last processed tuple and  $S_2 = \{\tau_2^2, \tau_2^3, \tau_2^1\}$ . Only the value of  $fmax(\tau_2^2) = 0.2$  is smaller than  $fmin(\tau_2^3) = 0.3$ . Therefore,  $M_2$  is defined by a pointer to  $\tau_2^2$  indicating that only  $\tau_2^2$  belongs to  $M_2$ .

The set of skyline tuples  $SKY(S_i)$  can also be maintained efficiently in order to speed-up the evaluation of the stopping condition. Again, due to the sorted access model, *SFSJ* will always append tuples in  $SKY(S_i)$ . Therefore, we only need to check whether a newly processed tuple  $\tau_i$  is dominated by a tuple  $\tau'_i \in SKY(S_i)$ . If this is not the case,  $\tau_i$  is inserted in  $SKY(S_i)$ .

Finally, to improve even further the performance of *SFSJ*, the tuple-set  $S_i$  can be maintained as the union of two sets of tuples namely  $SKY(S_i)$  and  $X_i = S_i - SKY(S_i)$ . A tuple  $\tau_i$  of  $S_i$  is only inserted in the set  $X_i$  if it is dominated by a tuple  $\tau'_i \in SKY(S_i) \cup B_i$ . In this way, the domination tests between join tuples are reduced, since the join results of a newly accessed tuple  $\tau_i$  with the tuples  $\tau_j \in SKY(S_j)$  can be added immediately to the skyline join result without any domination tests. Furthermore, if the newly accessed tuple  $\tau_i$  is a skyline tuple ( $\tau_i \in SKY(S_i)$ ), also the join tuples produced by joining  $\tau_i$  with tuples of  $X_j$  can be added immediately to the skyline join result without any domination tests. Both sets  $SKY(S_i)$  and  $X_i$  are implemented as sorted lists. Maintenance of tuple-set  $M_i$  is achieved by means of two pointers (for the sets  $SKY(S_i)$  and  $X_i$  respectively), which are managed as described previously.

## 6. ANALYSIS OF PULLING STRATEGIES

As mentioned in the previous section, *SFSJ* employs a pulling strategy to determine the order in which the two inputs are accessed. The choice of the strategy clearly affects how “deep” *SFSJ* reaches in its inputs, and hence it is interesting to investigate the existence of optimal strategies that minimize the number of accessed tuples, or equivalently, the amount of I/O performed by the algorithm. In what follows, we present a theoretical analysis of this problem and identify two provably good strategies: simple round-robin, and a novel adaptive strategy that we introduce below. The proofs of all theoretical results can be found in the Appendix.

**Adaptive Pulling.** The adaptive pulling strategy prioritizes access among the two relations based on the observed data. The main idea is to read tuples from a relation only if there is evidence that the new tuples will help satisfy the termination condition. Intuitively, this prioritization helps the algorithm terminate sooner, thus improving its performance.

Recall that the early termination condition requires each join value present in  $SKY(S_i)$  to be also present in  $MAD(S_j)$  ( $j \neq i$ ). Let us assume that the early termination condition does not hold, because there exists a tuple  $\tau_i \in SKY(S_i)$  for which  $\nexists \rho_j \in MAD(S_j)$  such that  $\tau_i.\alpha = \rho_j.\alpha$ . Note that  $SKY(S_i)$  can only grow as more tuples are accessed from  $S_i$ . Hence, the algorithm

can satisfy the early termination condition with respect to  $\tau_i \cdot \alpha$  only by enlarging  $MAD(S_j)$ , i.e., by pulling more tuples from  $R_j$ . Based on this property, we define the priority of  $R_j$  as the number of distinct mismatched join values. Formally:

**DEFINITION 5.** *The priority of a relation  $R_j$  is defined as  $priority_{SC}(R_j) = |\pi_\alpha(SKY(S_i)) - \pi_\alpha(MAD(S_j))|$ .*

The computation of  $priority_{SC}(R_j)$  can be folded in the evaluation of the termination condition (see Theorem 2). The adaptive pulling strategy always pulls from the relation with the highest priority. Ties are broken based on least current depth. The pulling strategy has two interesting properties. First,  $priority_{SC}(R_1) = priority_{SC}(R_2) = 0$  when the termination condition is satisfied. Second, the strategy will freeze access to an input if it cannot contribute to skyline results currently. As an example, consider again the two relations depicted in Figure 4. For all join values of the tuples in  $SKY(S_1)$ , there exists a tuple in  $MAD(S_2)$  with the same join value. Thus, tuples in  $R_2 - S_2$  combined with tuples of  $S_1$  cannot produce any skyline join result. The proposed pulling strategy will freeze access to relation  $R_2$  (since  $priority_{SC}(R_2) = 0$ ) and pull from  $R_1$ . No further tuples of  $R_2$  will be accessed, unless new join values are added to  $SKY(S_1)$ .

**Optimality Analysis.** We first describe the performance metrics that we consider and the associated notions of optimality. We write  $\mathcal{I}$  to denote the set of skyline join problem instances. We use  $\mathcal{SJA}$  to denote the class of deterministic algorithms that solve correctly any problem instance  $I \in \mathcal{I}$ , and write  $\mathcal{SFSJ}$  to denote the set of all instantiations of  $SFSJ$  with a different pulling strategy ( $\mathcal{SFSJ} \subseteq \mathcal{SJA}$ ). Given  $I \in \mathcal{I}$  and  $A \in \mathcal{SJA}$ , we define  $depth(A, I, i)$  as the number of tuples accessed from relation  $R_i$  when  $A$  terminates. We also define a total depth metric  $totDepth(A, I) = depth(A, I, 1) + depth(A, I, 2)$ . This metric is indicative of the total I/O performed by  $A$ , and hence it has a direct correspondence to the efficiency of the algorithm.

We employ two notions of optimality. Let  $\mathbb{A}$  be a class of skyline join algorithms. We say that an algorithm  $A \in \mathbb{A}$  is optimal if  $totDepth(A, I) \leq totDepth(B, I)$  for any instance  $I \in \mathcal{I}$  and algorithm  $B \in \mathbb{A}$ . We say that  $A$  is instance-optimal if there exist constants  $c$  and  $c'$  such that  $totDepth(A, I) \leq c \cdot totDepth(B, I) + c'$  for any instance  $I \in \mathcal{I}$  and algorithm  $B \in \mathbb{A}$ . Intuitively,  $A$  always performs within a factor of  $c$  of the optimal algorithm for any instance  $I$ . Constant  $c$  is termed the instance-optimality ratio of  $A$ .

In the first part of our analysis, we set  $\mathbb{A} = \mathcal{SFSJ}$  in order to identify optimal strategies for the  $SFSJ$  algorithm. We begin with an examination of the round-robin strategy. In what follows, we use  $SFSJ-RR$  to denote the corresponding instantiation of  $SFSJ$ . Our first result states that  $SFSJ-RR$  is instance optimal with a ratio of two. The significance of this finding is that  $SFSJ-RR$  will never perform arbitrarily bad for any problem instance, and thus carries a property of robustness.

**THEOREM 4.**  *$SFSJ-RR$  is instance-optimal within  $\mathcal{SFSJ}$  and  $\mathcal{I}$  with a ratio of 2.*

A natural question is whether we can identify an optimal pulling strategy. We answer this positively, showing that the  $SC$  pulling strategy (see Definition 5) is optimal. Formally, let  $SFSJ-SC$  be the instantiation of  $SFSJ$  with the  $SC$  strategy.

**THEOREM 5.**  *$SFSJ-SC$  is optimal within  $\mathcal{SFSJ}$  and  $\mathcal{I}$ .*

In fact, we can show an even stronger result: no other instantiation of  $SFSJ$  will outperform  $SFSJ-SC$  on any input relation.

Thus,  $SFSJ-SC$  is optimal in a stronger sense than implied with the  $totDepth$  metric.

The next part of our analysis considers the performance of pulling strategies within the larger class  $\mathcal{SJA}$ . The goal is now different: instead of comparing pulling strategies to each other, we basically want to compare  $SFSJ$  (with appropriate strategies) against other deterministic algorithms.

We develop our first result for a specific class of problem instances. We define  $\mathcal{I}^K$  as the class of problem instances such that at most  $K$  tuples in each input have the same  $fmin$  value. The reason we target this class is due to the access logic of  $SFSJ$ , which examines tuples in groups of the same  $fmin$  value. Class  $\mathcal{I}^K$  essentially bounds the size of these groups, and hence the amount of data that  $SFSJ$  must read before termination. In this case, we prove that  $SFSJ-RR$  and  $SFSJ-SC$  are instance optimal with a ratio of two.

**THEOREM 6.** *Both  $SFSJ-RR$  and  $SFSJ-SC$  are instance optimal within  $\mathcal{SJA}$  and  $\mathcal{I}^K$  with a ratio of 2.*

Up to this point, our results provide strong evidence for the choice of  $SFSJ-SC$  as the de jure instantiation of  $SFSJ$ .  $SFSJ-SC$  employs the optimal access strategy within the instantiations of  $SFSJ$ , and hence has the optimal I/O “footprint” in this family. Within the larger context of deterministic algorithms,  $SFSJ-SC$  may not be optimal but it maintains the important property of instance optimality when  $fmin$  values have bounded repetitions.

A favorable case can also be made for  $SFSJ-RR$ . The round-robin strategy is instance optimal in all examined cases, and is certainly more efficient than the adaptive pulling strategy which requires the manipulation of sets. Indeed, it is important to point out that all optimality results are with respect to the  $totDepth$  metric that reflects I/O performance, but the total running time of the algorithm also includes the computational overhead of the pulling strategy.

Can we show similar optimality results for the general class of inputs  $\mathcal{I}$ ? Unfortunately, the following theorem gives a negative answer.

**THEOREM 7.** *There exists no algorithm  $A \in \mathcal{SFSJ}$  such that  $A$  is instance-optimal within  $\mathcal{SJA}$  and  $\mathcal{I}$ .*

The problem is that  $SFSJ$  accesses tuples from each input in groups of the same  $fmin$  value, and the size of these groups can be unbounded in the general case. In turn, this precludes any performance guarantees. The investigation of an instance-optimal skyline join algorithm within  $\mathcal{SJA}$  and  $\mathcal{I}$  remains an interesting direction for future work.

## 7. EXPERIMENTAL EVALUATION

In this section, we perform an experimental evaluation of the  $SFSJ$  algorithm and we report our main findings. All algorithms are implemented in Java and the experiments run on a machine with an Intel Core2 Quad 2.33GHz processor and 4GB of RAM.

### 7.1 Experimental Setup

The basic experimental setup employs an instance of the skyline join problem where  $R_1 \bowtie R_2$  is a many-to-many join. The cardinality of each relation is denoted as  $|R_i|$ ,  $i = 1, 2$ . Each relation has  $m_i$  ( $i = 1, 2$ ) scoring attributes (i.e.,  $|B_i| = m_i$ ).

**Data sets and Queries.** We generated synthetic data sets in order to study the effect of different data characteristics on the performance of skyline join algorithms. As common in skyline research,

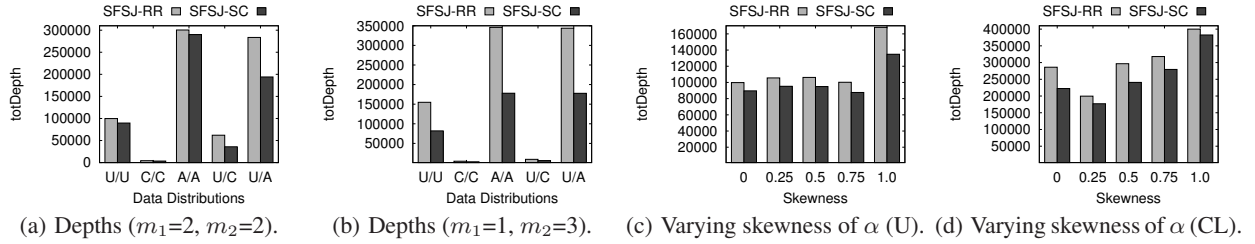


Figure 6: Comparison of pulling strategies for different data distributions.

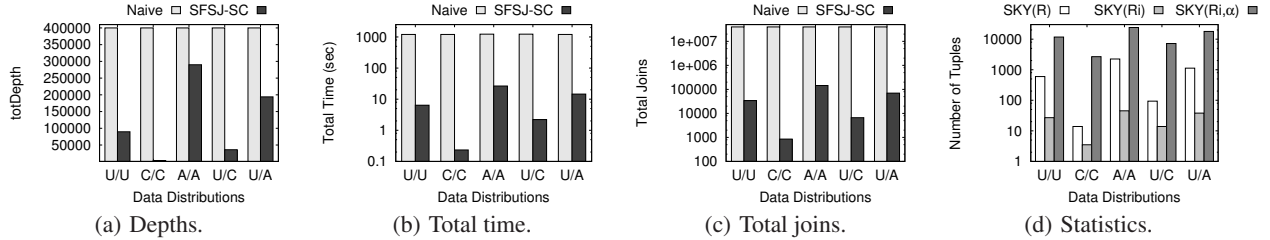


Figure 7: Comparison of *SFSJ-SC* to Naive for different data distributions.

we used uniform (U), correlated (C) and anti-correlated (A) data distribution for the scoring attributes. In addition, we used a clustered (CL) data distribution, where 5 cluster centroids are generated at random and the value of each scoring attribute follows a normal distribution with variance 0.05 and mean equal to the corresponding coordinate of the centroid. We also vary the distribution of join values using a zipfian distribution with parameter ranging from 0 (uniform) to 1 (skewed). Our default setup is:  $|R_i|=200K$ ,  $|m_i|=2$ , the join selectivity is  $10^{-3}$ , the zipf parameter is set to 0, and the data distribution is uniform. Notice that we conducted experiments with different join selectivities and obtained similar results.

In order to test the performance of *SFSJ* with real data, we used the NBA data set<sup>3</sup>, which contains statistics of NBA players from 1946 to 2008. We created two data sets termed *NBA-1* and *NBA-2*. In *NBA-1*, the relations  $R_1$  and  $R_2$  record performance statistics for players during the regular season on a yearly basis. Relation  $R_1$  records the field goals made and free throws, and relation  $R_2$  records the number of points and assists. Each relation contains 21384 tuples. The skyline join is performed on attributes {player,year}, and the query tries to identify the best pairs of players on a yearly basis. In *NBA-2*,  $R_2$  is the same as *NBA-1*, while  $R_1$  contains 1307 tuples and records the number of wins and the number of points made by each team in each season. The skyline join is performed on attributes {team,year} and its purpose is to retrieve the best teams and corresponding players on a yearly basis.

**Algorithms.** We implemented *SFSJ* with the round-robin (*SFSJ-RR*) and the adaptive pulling strategies (*SFSJ-SC*). We also implemented three competitor skyline join techniques. The first is an algorithm (denoted as *Naive*) that takes as input the initial relations and performs the complete join, while discarding dominated join tuples during processing. The second competitor is the approach of Jin et al. [6], denoted as *SMSJ* (sort-merge skyline join). We implemented the most efficient of the *SMSJ* variants, as reported in [6]. This variant first computes the skyline set and the group skylines of each relation ( $R_i$ ) and then processes each relation sorted ( $R'_i$ ) with skyline tuples preceding the group skyline tuples after having discarded the remaining tuples. For the necessary skyline computation, we employ the SFS algorithm [4]. *SFSJ* and *SMSJ* (through its usage of SFS) require their input to be sorted

by *fmin* and entropy value respectively. The third competitor is the *ProgXe* algorithm [10]. We implemented the version of *ProgXe* that employs progressive driven ordering, as reported in [10]. To ensure a fair comparison, the cost of preprocessing is discounted for all algorithms. Qualitatively, *ProgXe* has the highest overhead, due to the construction of a customized index structure. Both *SFSJ* and *SMSJ* have lower overhead, as they merely require sorting of the inputs. The Naive algorithm requires no preprocessing, but on the other hand it requires a full computation of the join results.

**Metrics.** Given a problem instance, we employ the following metrics: a) *totDepth* which is the sum of depths of accessed tuples for both relations, b) the total execution time, c) the number of dominance tests, and d) the total number of generated join results that each algorithm performed for producing the skyline join result.

For our experiments on synthetic data, we report the average of each metric over 10 different instances of the data set. We generate the different instances by keeping the parameters fixed and changing the seeds of the random number generator. We adopt this approach in order to factor out the effects of randomization.

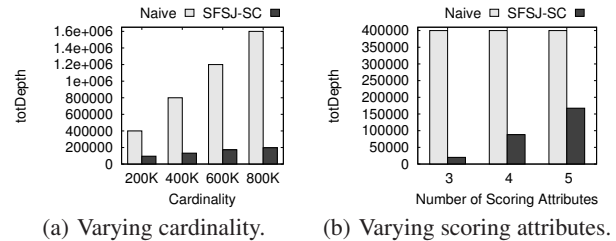


Figure 8: Sensitivity analysis of algorithm *SFSJ-SC*.

## 7.2 Experimental Results

**Evaluation of Pulling Strategies.** In Figure 6, we perform an experiment that aims to quantify the gain of *SFSJ-SC* compared to *SFSJ-RR*. Our goal is to gather empirical evidence to complement our theoretical analysis that *SFSJ-SC* always terminates earlier than any other pulling strategy.

In Figure 6(a), we use the default setup of two scoring attributes per relation ( $m_1=2$  and  $m_2=2$ ). In all cases, *SFSJ-SC* performs better than its competitor and stops at a lower depth. We performed the same experiment for  $m_1=1$  and  $m_2=3$  in order to examine the

<sup>3</sup>Available from: <http://databasebasketball.com>.

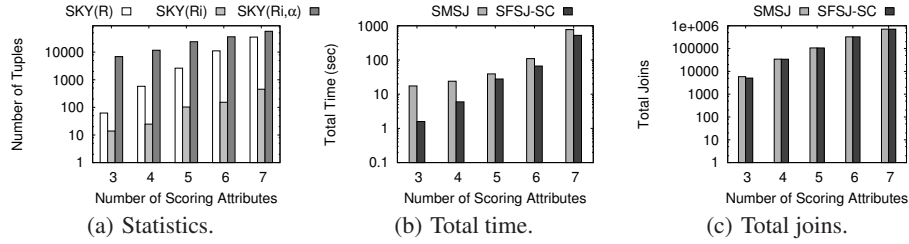


Figure 9: Comparison of *SFSJ* to *SMSJ* algorithm.

effect of different number of scoring attributes per relation. Figure 6(b) shows the results. Again, *SFSJ-SC* performs much better than *SFSJ-RR* in all setups. It is noteworthy that the increased number of scoring attributes in  $R_2$  for *A/A* and *U/A* data distributions makes *SFSJ-RR* read more tuples in total. In contrast, *SFSJ-SC* is more robust and is not affected by this setup. We also provide in Figure 6(c) an experiment for evaluating the effect of varying the skewness in the join values. So far, we examined the case that the join values are uniformly distributed (skewness 0). We gradually increase the skewness using a zipf distribution with parameter up to 1. The chart shows that *SFSJ-SC* consistently outperforms the round-robin pulling strategy, irrespective of skewness in the join values. The measurements for the total time metric followed the same trends as the *totDepth* metric, and are omitted in the interest of space. Similar results are obtained in Figure 6(d) for the clustered data set, only the absolute values are higher, because any algorithm needs to examine more tuples to compute the skyline join result.

**Comparison to Naive.** Next, we study the performance of *SFSJ-SC* compared to the Naive algorithm for all tested data distributions. Figure 7 depicts the results of our comparison. With respect to the *totDepth* metric (Figure 7(a)), notice that the Naive algorithm always needs to access the complete inputs, therefore its *totDepth* equals to  $|R_1| + |R_2|$ . *SFSJ-SC* always terminates much earlier than Naive, and in the best case (*C/C*) only accesses less than 1% of the *totDepth* of Naive. For the demanding *A/A* data distribution, where any algorithm is prone to go deep to ensure safe termination, *SFSJ-SC* needs to access more tuples, but still improves Naive by more than 25%. In terms of total time, *SFSJ-SC* is notably more efficient than Naive, as shown in Figure 7(b). Even in the case of *A/A* data set, *SFSJ-SC* is almost two orders of magnitude faster than Naive. The reason is clearly depicted in Figure 7(c) that shows the number of computed join results for each algorithm. *SFSJ-SC* performs significantly fewer join operations than Naive, because *SFSJ-SC* a) avoids many unnecessary join operations by eagerly pruning tuples that cannot produce skyline join results, and b) exploits the early termination condition to stop accessing the input relations as soon as the complete skyline join result is computed.

In Figure 7(d), we provide some interesting statistics that are relevant to the previous performance results. Specifically, we report the number of: skyline join results ( $SKY(R)$ ), skyline tuples per relation ( $SKY(R_i)$ ), and group skyline tuples per relation ( $SKY(R_i, \alpha)$ ). Notice that the performance of *SFSJ-SC* in terms of total time and total joins relates directly to the cardinality of  $SKY(R)$ . Except for the *U/C* and *C/C* data sets, we note that the cardinality of  $SKY(R)$  is high (close to 1000), with highest values for data set *A/A*. Also, we observe that  $SKY(R_i, \alpha)$  is at least one order of magnitude larger than  $SKY(R)$ .

**Sensitivity Analysis.** The next set of experiments considers the sensitivity of *SFSJ* with respect to several interesting parameters. We also include *Naive* as a baseline. In the interest of space, we

only show the results for the *totDepth* metric, and report on the trends of the other two metrics.

We first consider the effect of input cardinality as it increases from 200K to 800K tuples. The results for the two algorithms are shown in Figure 8(a). *SFSJ-SC* always terminates at an earlier depth than Naive. In terms of total time, *SFSJ-SC* is consistently more than two orders of magnitude faster than Naive and scales gracefully even when the cardinality increases by a factor of 4. The same result is obtained for the number of total join operations performed by each algorithm to produce the skyline join result. *SFSJ-SC* is highly efficient and is practically unaffected by the increased cardinality, whereas Naive scales poorly. Figure 8(b) studies the effect of increasing the number of scoring attributes. *SFSJ-SC* terminates earlier than Naive in all setups. An increased number of scoring attributes causes *SFSJ-SC* to scan more tuples, which is expected as the skyline and group skyline tuples increase rapidly with increasing dimensionality.

**Comparative Study to *SMSJ*.** Figure 9 presents the results of a comparative study between *SFSJ-SC* and the *SMSJ* algorithm [6]. We study the scalability of both algorithms for increased number of scoring attributes, from 3 to 7. We do not report results on the *totDepth* metric, since *SMSJ* computes the skyline over relations that result from a transformation of the original relations.

Figure 9(a) shows the number of skyline join results, skyline and group skyline tuples in the data set, in order to aid the interpretation of the experimental results. All values increase with the number of scoring attributes. Notice that the number of skyline join results increases significantly as the number of scoring attributes grows. In Figure 9(b), we measure the total time of *SFSJ-SC* versus *SMSJ*. Our algorithm is faster than *SMSJ*, due to the non-negligible processing cost of *SMSJ* for computing the skyline and group skyline tuples. This is clearly demonstrated in Figure 9(c), where the number of total join operations that each algorithm performs is shown. Notice that *SFSJ-SC* requires slightly fewer join operations than *SMSJ*, although *SMSJ* operates only on the skyline and group skyline tuples of each relation. This is a strong argument in favor of a) the online pruning achieved by *SFSJ* due to sorted access, and b) the efficiency of the stopping condition that avoids redundant accesses on the input relations.

**Comparative Study to *ProgXe*.** In Figure 10, we compare the performance of *SFSJ* to *ProgXe* [10] for different data distributions. We use 20K tuples per relation and  $m_1=m_2=2$  scoring attributes. We set the number of input and output partitions per dimension to 10 and 20 respectively. We note that there is no way to tune these parameters, unless one evaluates all possible values. Figure 10(a) shows that in all tested data distributions *ProgXe* needs to exhaust both inputs to ensure that the complete and correct result set is reported. Although *ProgXe* avoids processing all possible combinations of input partitions, it still accesses each input partition at least once, hence it cannot terminate early (in contrast to *SFSJ*). Figure 10(b) depicts the number of dominance tests employed by each algorithm. Clearly, *ProgXe* performs significantly more dom-

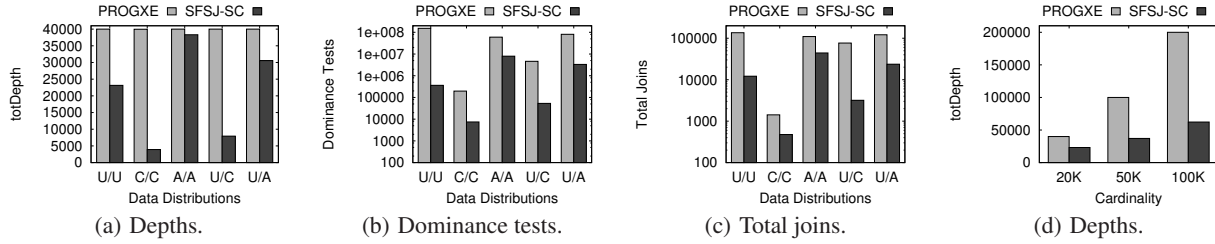


Figure 10: Comparison of *SFSJ* to *ProgXe* algorithm.

inance tests than *SFSJ*, because *ProgXe* uses dominance tests not only between tuples but also between partitions. Also, as shown in Figure 10(c), *ProgXe* needs to perform more join operations of tuples to produce the skyline join result. Then, in Figure 10(d), we increase the cardinality of the input relations. The results show that the relative gain of *SFSJ* to *ProgXe* increases with the cardinality of the relations, demonstrating the scalability of our approach.

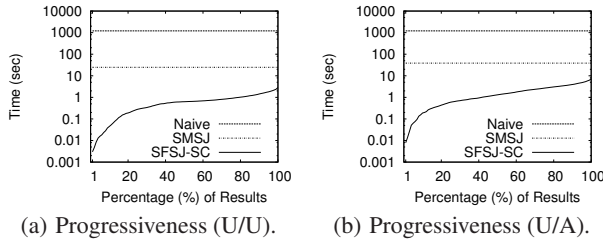


Figure 11: Progressiveness.

**Progressiveness.** Furthermore, in the next set of experiments in Figures 11(a) and 11(b), we evaluate the progressive property of *SFSJ* algorithm. The charts correspond to the data distributions U/U and U/A respectively, in order to examine our default setup and a hard setup for any algorithm, while the U/C and C/C setups are trivial. The y-axis shows the time at which a percentage (shown on the x-axis) of the total skyline join result set is reported. In all cases, *SFSJ-SC* reports a significant percentage of the result set in a fraction of a second. The charts clearly demonstrate the advantages of *SFSJ-SC*, in contrast to *SMSJ* and Naive which need to complete processing, before reporting the skyline join result.

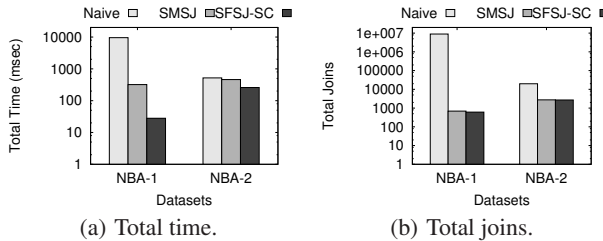


Figure 12: Real data sets (NBA-1 and NBA-2).

**Experiments with Real Data.** In Figure 12, we report the experimental results on the real data sets. The general trends are in accordance with the study of the synthetic data sets. Notice that the absolute measured values are smaller than in the case of the synthetic data, because the real data set is smaller than the synthetic ones. The total time for both data sets (NBA-1 and NBA-2) is shown in Figure 12(a), where *SFSJ-SC* outperforms the other algorithms in both cases. Also, in Figure 12(b), we depict the number of join operations performed, and *SFSJ-SC* requires marginally fewer operations than *SMSJ* and significantly fewer than Naive. In all cases, *SFSJ-SC* outperforms its competitors for both data sets.

## 8. CONCLUSIONS

In this paper, we study the problem of computing the skyline set of a join assuming that the join inputs are accessed in sorted order. We introduced the *SFSJ* algorithm that combines several nice properties: it avoids useless join computation; it generates the results progressively; it can terminate early; and, it has strong optimality guarantees for its performance. The experimental results validated the effectiveness of *SFSJ* and demonstrated its numerous advantages over competitor techniques.

**Acknowledgments.** This work was supported in part by the National Science Foundation under Grant No. IIS-0447966.

## References

- [1] I. Bartolini, P. Ciaccia, and M. Patella. SaLSa: computing the skyline without scanning the whole sky. In *Proceedings of CIKM*, 2006.
- [2] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *ACM Trans. Database Syst.*, 33(4), 2008.
- [3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of ICDE*, pages 421–430, 2001.
- [4] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proceedings of ICDE*, pages 717–816, 2003.
- [5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of PODS*, 2001.
- [6] W. Jin, M. Ester, Z. Hu, and J. Han. The multi-relational skyline operator. In *Proceedings of ICDE*, 2007.
- [7] W. Jin, M. Morse, J. Patel, M. Ester, and Z. Hu. Evaluating skylines in the presence of equi-joins. In *Proceedings of ICDE*, 2010.
- [8] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *Proceedings of VLDB*, 2006.
- [9] J. Levandoski, M. F. Mokbel, and M. Khalefa. FlexPref: A framework for extensible preference evaluation in database systems. In *Proceedings of ICDE*, 2010.
- [10] V. Raghavan and E. Rundensteiner. Progressive result generation for multi-criteria decision support queries. In *Proceedings of ICDE*, 2010.
- [11] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *Proceedings of PODS*, 2008.
- [12] D. Sun, S. Wu, J. Li, and A. K. H. Tung. Skyline-join in distributed databases. In *ICDE Workshops*, 2008.
- [13] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *Proceedings of ICDE*, 2006.
- [14] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. *PVLDB*, 2(1):898–909, 2009.

## APPENDIX

### A. PROOFS OF RESULTS IN SECTION 6

Our proofs make use of the following implications that are trivial to prove:  $\tau_i \in MAD(S_i) \Rightarrow \forall \rho_i \in R_i - S_i (\tau_i \prec \rho_i)$  and  $MAD(S_i) \neq \emptyset \Rightarrow SKY(S_i) = SKY(R_i)$ . We also employ the notation  $\hat{i}$  to denote the input index  $\{1, 2\} - \{i\}$  for  $i \in \{1, 2\}$ .

We begin by proving three auxiliary lemmata.

LEMMA 3.  $\max_{j=1,2} depth(SFSJ-RR, I, j) \leq \max_{j=1,2} depth(A, I, j)$   
 for all  $I \in \mathcal{I}$  and  $A \in SFSJ$ .

PROOF. By contradiction. Define  $d_k^A = \max_{j=1,2} \text{depth}(A, I, j)$  and assume that  $d_k < \max_{j=1,2} \text{depth}(SFSJ-RR, I, j)$ . It follows that  $SFSJ-RR$  pulls at depth  $d_k^A + 1$  on one input  $i$ . Let  $p_1, p_2$  be the depths of  $SFSJ-RR$  just before this happens (hence,  $p_i = d_k^A$ ), and let  $S_1, S_2$  denote the corresponding tuple-sets. Define  $S_1^A, S_2^A$  as the tuple-sets that  $A$  has examined when it terminates. It holds that  $d_k^A \leq d_k^A = p_i \leq p_i$ , where the last inequality follows from the round-robin strategy. We conclude that  $A$  has examined a subset of tuples observed by  $SFSJ-RR$ .

We know that  $A$  satisfies the stopping condition and hence  $MAD(S_j^A) \neq \emptyset$ . Given that  $S_j \supseteq S_j^A \Rightarrow MAD(S_j) \supseteq MAD(S_j^A) \neq \emptyset$ , we can assert that  $SKY(S_j) = SKY(R_j) = SKY(S_j^A)$ . The following implication follows immediately:

$$\begin{aligned} \pi_\alpha(SKY(S_j^A)) - \pi_\alpha(MAD(S_j^A)) &= \emptyset \Rightarrow \\ \pi_\alpha(SKY(S_j)) - \pi_\alpha(MAD(S_j)) &= \emptyset \end{aligned}$$

We can thus assert that the stopping condition must also be true for  $S_1$  and  $S_2$ , which contradicts our assumption that  $SFSJ-RR$  pulls past  $p_i$ .  $\square$

LEMMA 4.  $\text{depth}(SFSJ-SC, I, j) \leq \text{depth}(A, I, j)$  for all  $I \in \mathcal{I}$ ,  $A \in \mathcal{SFSJ}$  and  $j = 1, 2$ .

PROOF. By contradiction. Let us assume that  $A$  stops earlier on input  $k$ . Define  $d_k^A = \text{depth}(A, I, j)$ , and let  $S_1^A, S_2^A$  be the observed tuple-sets when  $A$  terminates. Clearly, the stopping condition is true for  $S_1^A$  and  $S_2^A$ .

Let  $S_1, S_2$  be the observed tuples for  $SFSJ-SC$  just before it pulls at depth  $d_k^A + 1$ . We have that  $S_k = S_k^A$ , and it must also hold that  $S_{\hat{k}} \subset S_{\hat{k}}^A$ , otherwise the stopping condition would be true for  $SFSJ-SC$  as well. Hence,  $SKY(S_{\hat{k}}) \subseteq SKY(S_{\hat{k}}^A)$ .

Since the stopping condition is satisfied when  $A$  terminates, it follows that  $\pi_\alpha(SKY(S_j^A)) - \pi_\alpha(MAD(S_j^A)) = \emptyset$ , for  $j = 1, 2$ . Given that  $SKY(S_k^A) \supseteq SKY(S_{\hat{k}})$  and  $MAD(S_k) = MAD(S_k^A)$ , we conclude that  $\pi_\alpha(SKY(S_{\hat{k}})) - \pi_\alpha(MAD(S_k)) = \emptyset$ . This implies that  $\text{priority}_{SC}(R_k) = 0$ . However, since  $\text{priority}_{SC}(R_k) \geq \text{priority}_{SC}(R_{\hat{k}}) \geq 0$  ( $SFSJ-SC$  pulls from relation  $R_k$  next), it must hold that  $\text{priority}_{SC}(R_{\hat{k}}) = 0$ , which in turn means that the stopping condition is satisfied with  $S_1$  and  $S_2$ . This contradicts our assumption that  $A$  pulls past depth  $d_k^A + 1$ .  $\square$

LEMMA 5.  $\max_{j=1,2} \text{depth}(SFSJ-RR, I, j) \leq \max_{j=1,2} \text{depth}(A, I, j) + K$  for all  $I \in \mathcal{I}^K$  and  $A \in \mathcal{SJA}$ .

PROOF. Let  $k$  be the input index on which  $A$  pulls the most tuples. Let  $m$  be defined similarly for  $SFSJ-RR$ . We will assume that  $\text{depth}(SFSJ-RR, I, m) > \text{depth}(A, I, k) + K$  and show that this leads to a contradiction.

Let  $p_1$  and  $p_2$  be the depths of  $SFSJ-RR$  right before it pulls past depth  $\text{depth}(A, I, k) + K$  on input  $m$ . Let  $S_1$  and  $S_2$  denote the corresponding sets of seen tuples. Define  $d_1^A, d_2^A$  as the termination depths of  $A$ . The following is true:  $p_m = d_k^A + K \geq d_m^A + K$ , and  $p_m \geq p_m$ , due to the round-robin strategy. We conclude that  $p_m \geq d_m^A + K$ . Since both  $p_1$  and  $p_2$  exceed  $K$ , it means that  $SFSJ-RR$  has witnessed a change of  $\text{fmin}$  value in both inputs, and hence  $S_1 \neq \emptyset$  and  $S_2 \neq \emptyset$ .

Sets  $S_1$  and  $S_2$  do not satisfy the stopping condition, since  $SFSJ-RR$  pulls another tuple. This gives rise to two cases. In each case, we will construct a new instance  $I'$  on which  $A$  commits a mistake. This yields the contradiction that completes the proof, as we have assumed that  $A$  is a correct algorithm.

Case 1:  $\pi_\alpha(SKY(S_{\hat{m}})) - \pi_\alpha(MAD(S_{\hat{m}})) \neq \emptyset$  Since  $SKY(S_{\hat{m}}) \neq \emptyset$ , there exists a tuple  $\tau_{\hat{m}} \in SKY(S_{\hat{m}})$  such that  $\tau_{\hat{m}}.\alpha \equiv y \notin$

$\pi_\alpha(MAD(S_m))$ . Let  $\tau_m$  denote the  $(d_k^A) + 1$ -th tuple of  $R_m$ . We construct the new instance  $I'$  from  $I$  by substituting  $\tau_m$  with a new tuple  $\tau'_m$  such that:  $\tau'_m.\alpha = y$ , and all skyline coordinates of  $\tau'_m$  are set equal to  $\text{fmin}(\tau_m)$ . It is straightforward to verify that  $\tau'_m$  appears in  $SKY(R'_m, y)$ , otherwise the tuple that dominates it would be contained in  $MAD(S_m)$  and hence the value  $y$  would appear in  $\pi_\alpha(MAD(S_m))$ . We conclude that  $\tau_{\hat{m}} \bowtie \tau'_m$  must appear in the solution for  $I'$ , since  $\tau_{\hat{m}}$  appears in the global skyline and  $\tau'_m$  in a group skyline.

Case 2:  $\pi_\alpha(SKY(S_m)) - \pi_\alpha(MAD(S_{\hat{m}})) \neq \emptyset$  Following the same reasoning as before, we construct  $I'$  by altering the  $(d_k^A + 1)$ -th tuple in  $R_{\hat{m}}$ .

In both cases,  $I'$  changes the tuple at depth  $d_k^A + 1$  in one of the two inputs. This tuple is never accessed by  $A$  and also leads to the generation of a new result. It follows that  $A$  will compute the wrong solution.  $\square$

We now proceed with the proofs of the main theorems.

**Proof of Theorem 4** We use Lemma 3. Let  $d_1, d_2$  be the termination depths of  $SFSJ-RR$  and define  $d_1^A$  and  $d_2^A$  similarly for  $A$ .

$$\begin{aligned} \text{totDepth}(SFSJ-RR, I) &= d_1 + d_2 \leq 2 \cdot \max\{d_1, d_2\} \leq \\ &2 \cdot \max\{d_1^A, d_2^A\} \leq 2 \cdot (d_1^A + d_2^A) = 2 \cdot \text{totDepth}(A, I) \end{aligned}$$

$\square$

**Proof of Theorem 5** The proof follows directly from Lemma 4.  $\square$

**Proof of Theorem 6** We use Lemma 5. Let  $d_1$  and  $d_2$  denote the termination depths of  $SFSJ-RR$ , and define  $d_1^A$  and  $d_2^A$  similarly for  $A$ .

$$\begin{aligned} \text{totDepth}(SFSJ-RR, I) &= d_1 + d_2 \leq 2 \max\{d_1, d_2\} \\ &\leq 2 \max\{d_1^A, d_2^A\} + 2K \\ &\leq 2 \cdot \text{totDepth}(A, I) + 2K \end{aligned}$$

This proves that  $SFSJ-RR$  is instance optimal, since  $2K$  can be treated as a constant that does not depend on  $A$  or  $I$ . The instance optimality of  $SFSJ-SC$  follows directly by Lemma 4  $\square$

**Proof of Theorem 7** Consider the following instance in  $\mathcal{I}$ .

	$R_1$			$R_2$		
	$\alpha$	$\beta_1$	$\beta_2$	$\alpha$	$\beta_3$	
$\tau_1^1$	a	.2	.1	$\tau_2^1$	a	.1
$\tau_1^2$	a	.1	.2	$\tau_2^2$	b	.2
$\tau_1^3$	b	.3	.3			
...						
$\tau_1^{n+3}$	b	.3	.3			
$\tau_1^{n+4}$	b	.5	.5			

Suppose that an algorithm  $B \in \mathcal{SJA}$  accesses the following prefixes:  $S'_1 = \{\tau_1^1, \tau_1^2, \tau_1^3\}$  from  $R_1$  and  $S'_2 = \{\tau_2^1, \tau_2^2\}$  from  $R_2$ . Based on the sorted access model,  $B$  can reason that  $SKY(R_1) = \{\tau_1^1, \tau_1^2\}$ , since  $\tau_1^3$  will dominate any other tuple in  $R_1 - S'_1$ . Similarly, it is straightforward to verify that  $SKY(R_2) = \{\tau_2^1\}$ . Since  $SKY(R_1) \bowtie SKY(R_2) \equiv SKY(R_1) \times SKY(R_2)$  for this case,  $B$  can safely terminate since there are no more skyline results.

Consider an algorithm  $A \in \mathcal{SJA}$ . In order to make  $MAD(S_1) \neq \emptyset$ , and thus have a chance to satisfy the stopping condition, the algorithm has to access at least until  $\tau_1^{4+n}$ . Hence,  $\text{totDepth}(A, I)$  can become arbitrarily worse compared to  $\text{totDepth}(B, I) = 5$ , which in turn implies that  $A$  cannot be instance-optimal.  $\square$