Evaluating Skylines in the Presence of Equijoins

Wen Jin\(^1\), Michael D. Morse\(^1\), Jignesh M. Patel\(^2\), Martin Ester\(^3\), Zengjian Hu\(^3\)

\(^1\)University of Michigan
\{wenjin, mmorse\}@eecs.umich.edu

\(^2\)University of Wisconsin-Madison
jignesh@cs.wisc.edu

\(^3\)Simon Fraser University
{ester, zhu}@cs.sfu.ca

Abstract—When a database system is extended with the skyline operator, it is important to determine the most efficient way to execute a skyline query across tables with join operations. This paper describes a framework for evaluating Skylines in the presence of equijoins, including: (1) the development of algorithms to answer such queries over large input tables in a non-blocking, pipeline fashion, which significantly speeds up the entire query evaluation time. These algorithms are built on top of the traditional relational Nested-Loop and the Sort-Merge join algorithms, which allows easy implementation of these methods in existing relational systems; (2) a novel method for estimating the skyline selectivity of the joined table; (3) evaluation of skyline computation based on the estimation method and the proposed evaluation techniques; and (4) a systematic experimental evaluation to validate our skyline evaluation framework.

I. INTRODUCTION

Skyline evaluation has recently emerged as an important operator in many important decision support and visualization applications and has also attracted significant interest in the database research community. Bőrzsönyi et al. [1] first proposed to extend existing database systems with the skyline operator, and triggered the development of many algorithms for computing the skyline and its variants [1], [9], [10]. Borzsonyi et al. [1] proposes to reduce the size of the input table to determine the most efficient way to execute a skyline query over equijoin outputs.

The method to efficiently answer such a query over Players and Teams is different from that of answering a “skyline player” query only on the Players table, or a “skyline team” query only on the Teams table. Essentially we need to examine the interactions between the skyline and the join operators and design a combined and efficient evaluation scheme. For example, if we start the skyline operation after the join operation, then the total cost of the query is the sum of the skyline computation and the join cost. Additionally, this sequential execution causes the skyline query to be blocked by the join operator. Furthermore, skyline algorithms that use indices constructed on the skyline attributes (e.g. BBS [7]) have to pay the cost of building an index on the fly on the join results, which could be very expensive. In contrast, a better alternative is to pipeline the join results to the skyline operator, allowing overlap in the computation of these two operations and also producing skyline results progressively.

In general, given any two tables \(R \) and \(S \), many factors can affect the performance of a skyline query over \( R \Join S \). These factors include the size of \( R \) and \( S \), the size of \( R \Join S \), the skyline cardinality of \( R \Join S \), the selection of the join algorithms and the skyline algorithms, and the adaptation of these algorithms to the skyline computation. In addition, since skyline computation is CPU intensive, we cannot simply ignore its CPU costs and have to consider it along with the I/O costs. Consequently, we need to build an evaluation framework to determine the most efficient way to execute a skyline query over equijoin outputs.

Very few of the previous work on skyline evaluation consider Skylines in the presence of joins [1], [9], [10]. Borzsonyi et al. [1] proposes to reduce the size of the input table.
by computing local skylines in the grouping tuples, and in particular discuss a special case where the skyline attributes in the joined table are all from one input table. Sun et al. [10] considers a distributed scenario and proposes two schemes to handle skyline join operations, where one scheme is the extension of an existing skyline algorithm (SaLSa [11]), and the other one is an iterative algorithm. The work in [9] assumes the “non-reductive” property, which states that the size of the joined table is always larger than the input tables, and presents methods relying on the pre-computed skylines from both input tables. None of these methods considers the relationship between the join and the skyline operators, nor provides any skyline query evaluation framework in the general case.

This paper makes the following contributions:

- We propose a framework for evaluating queries that involve skylines computed over an equijoin between two tables.
- We develop algorithms to compute the answer to such queries over large input tables in a non-blocking, pipelined fashion, which speeds up the overall query evaluation significantly. These algorithms use traditional Nested-Loop and Sort-Merge join algorithms as building blocks, thereby allowing easy implementation of these methods in existing relational DBMSs;
- We present a novel method to estimate the size of skylines in join result, and formulate cost analyses for each skyline evaluation method.
- We present a systematic evaluation of our methods on both synthetic and real datasets, validating the effectiveness of our evaluation framework.

The remainder of this paper is organized as follows: Section II surveys related work, and Section III introduces the methods for computing skylines in the presence of equijoins. Section IV presents our skyline cardinality estimation method and discusses various evaluation plans for skyline queries. Section V provides experimental results, and Section VI contains our concluding remarks.

II. RELATED WORK

Given a d-dimensional dataset, a skyline query retrieves all tuples that are not dominated by any other tuple. Here, a tuple p dominates another tuple p’ when p is at least as preferable to p’ in all dimensions and more preferable to p’ in at least one dimension, for a defined preference function.

Recent studies on the skyline operator in the database community can be categorized as follows:

Generic Approaches: The generic algorithms require no expensive pre-processing or index structure to be built on the dataset, but need to scan the entire dataset. The first two generic algorithms for skyline computation in the relational context are block-nested loops (BNL) and divide-and-conquer (D&kC) [1]. Chomicki et al. proposed a Sort-First-Skyline (SFS) [3], an improved version of BNL, which requires the dataset to be initially sorted by entropy values. Processing the sorted data has the advantage that no point can be dominated by any point that comes after it, thus saving on the number of tuple comparisons. Godfrey et al. proposed the Linear Elimination Sort For Skyline method (LESS) based on SFS, with average complexity $O(d \cdot n)$ (d is the dimensionality and n is the number of tuples). This method is an important component in our evaluation framework, and we discuss its details in Section III.

Index-based Approaches: Index-based skyline algorithms do not need to scan the whole dataset and, hence, can obtain sub-linear performance, producing skyline points progressively. Tan et al. [8] proposed a Bitmap approach and an Index approach to progressively compute skylines. Kossmann et al. present the Shooting-Star algorithm to produce skylines by using the nearest neighbor search in R-trees to partition the space recursively [5]. Papadias et al. [7] improved their work by applying a Branch-and-Bound Skyline (BBS) technique based on the best-first search in R-trees to produce an I/O optimal algorithm.

Lin et al. [12] proposed finding the k most dominating tuples in a dataset and developed methods to compute such tuples using R-trees. The computed tuples can potentially improve the pruning capabilities of LESS, but R-trees typically do not perform well on high dimensional data. Their algorithm is also not progressive.

Joined Table Skyline Approaches Borzsonyi et al. [1] proposed to build the skyline operator on top of the join operator, and discussed a special case where skyline attributes in the joined table are all from one input table. Sun et al. [10] proposes two methods to handle skyline join operations: one is the extension of an existing skyline algorithm, SaLSa [11], and the other one prunes the search space iteratively. The work by Jin et al. [9] assumed that the non-reductive property holds for the join operation, i.e. the size of the joined table is larger or equal to that of any table participating in the join operation, and their methods relied on the pre-computation of skylines in each input table.

III. COMPUTING SKYLINES IN THE PRESENCE OF EQUIJOINS

In this section, we first present a generic query plan for the evaluation of skylines when tables must first be joined. Any combination of existing join and skyline techniques can be applied to this generic query plan to produce the resulting
skyline. We then discuss four algorithms for skyline evaluation that replace the join and skyline operators in this generic query plan to improve performance.

A. Generic Query Plan for Skyline Joins

The generic query plan is shown in Figure 1(a). It consists of two operators: the join operator and the skyline operator. We restrict ourselves to joins involving two tables, R and S, but the basic query plan can be easily extended for any arbitrary number of input tables. We further assume (1) \(|R| \leq |S|\), (2) skyline attributes are of the same data type, and (3) the equi-join attribute of the two relations does not take part in the skyline evaluation (assuming without loss of generality only one join attribute from R and S each).

In the general case depicted in Figure 1(a), the R and S relations can be joined together using any technique to produce an intermediate relation that can then be processed by any existing skyline algorithm. This is the current state-of-the-art, and in the rest of this paper we develop methods that improve the performance of the entire query over this naive case. For simplicity, we assume no other operation such as selection, projection, aggregation, etc. in the skyline query (however it is easy to extend these operations to the framework). For reference, common notations used in this paper are listed in Table I.

We choose to develop our techniques around the (basic and index) Nested-Loop Join (NLJ) and Sort-Merge Join (SMJ) operators. We have selected these two because they are common join algorithms, and a fair amount of skyline processing can be interwoven into the workings of these join algorithms. For example, in the case of SMJ, pre-sorting the two relations is an operation that can be advantageous for skyline computation.

We augment these two join operators by incorporating methods from the SFS and LESS algorithms. We have selected these two algorithms because of their average case linear complexity, the fact that they do not require the construction of indices, their applicability on datasets with higher dimensionality (which limits index-based techniques), and because they are the best general-case skyline algorithms.

SFS [3] operates by first sorting \(n\) tuples using an entropy function \(Ent(t) = \sum_{i=1}^{d} (\ln t'(a_i) + 1)\), where \(t(a_i)\) is value of a tuple \(t\) on dimension \(a_i\), and \(t'(a_i)\) is the normalized value of \(t(a_i)\). This entropy function guarantees that tuples cannot be dominated by those that occur after them in the sort order. SFS then computes the skyline by comparing each point with the skyline points found so far. The average cost of SFS after sorting is \(O(d \cdot n)\).

LESS is an extension of SFS, with two major improvements. A block diagram that outlines the steps in the LESS algorithm is shown in Figure 1(b). LESS uses an Elimination-filter (EF) window in the first pass of the external sorting process to quickly eliminate some dominated tuples. During the sorting, tuples with the best entropy scores seen so far are kept in the EF window. Each time that a block of tuples is read for sorting, they are compared with the tuples in the EF window first. Tuples that are dominated by a data point in the EF window can be discarded from further consideration. LESS is thus able to reduce the number of tuples in the first sorting pass. LESS also combines the final external sorting pass with the first skyline elimination pass. This saves a pass through the data for computing skylines if there are less than \(B-2\) runs in the last pass, where \(B\) is the number of buffer pages. If the buffer size is large enough to hold the tuples for initial sorting after filtering, then the sorted tuples can be directly processed by SFS.

B. Basic Cost Analysis

In the following sections, the Sort-Merge Join (SMJ), Nested-Loops Join (NLJ), Index Nested-Loops Join (INLJ), and LESS (external sorting plus SFS) algorithms are used as the basic building blocks for more complex evaluation plans.

We look at I/O cost first. Without loss of generality, for relation \(R\) with cardinality \(|R|\) and dimensionality \(d_R\), the total number of pages, \(|R|\), is \(|R| = \frac{|R| \cdot |R|}{b_R \cdot c_R}\).

The number of I/O incurred when sorting \(R\) [13] is:

\[
XSort_{IO}(R) = |R| \log_{B1-1} |R|
\]

The number of I/O incurred by SMJ when merging \(R\) and \(S\) after sorting [14] is: \(SMJ_{IO}(R, S) = |R| + |S|\) [15]. Similarly, the number of page accesses in Block Nested Loop Join (BNLJ) and Index Nested Loop Join (INDJ) when joining \(R\) and \(S\) is as follows [13], [14]:

\[
BNLJ_{IO}(R, S) = \frac{|R| \cdot |R|}{B_R} + |R| + |S|
\]

\[
INLJ_{IO}(R, S) = |R| + |R| \cdot c
\]

where \(c\) is a matching constant (typically \(c\) is small).

Let the number of tuples from \(R \times S\) passed to SFS be \(n_E\) (note these tuples are after pruning). Then, the number of
skyline tuples produced is \(|(R \bowtie S)_{sky}|\), and the number of page accesses required in \(SFS\) is (as it has a linear cost [4], so \(\frac{|(R \bowtie S)_{sky}|}{R_2 \cdot B_2}\) rounds of processing needs to be done):

\[
SFS_{10}(n_F, (R \bowtie S)_{sky}) = n_F \cdot \frac{|(R \bowtie S)_{sky}|}{B_2}
\]

The CPU cost for sorting \(R\) is [13], [14]:

\[
XSort_{cpu}(R) = \frac{|R| \cdot B_s}{d_R} \cdot \log \frac{B_s \cdot B_1}{d_R}
\]

The CPU cost of comparing and merging records in sorted runs of relation \(R\) during join [13], [14] is:

\[
Merge_{cpu}(R) = \frac{|R| \cdot \log B_{1-1}}{|R|}
\]

The CPU cost for \(SFS\) [4] processing \(n_F\) joined records is:

\[
SFS_{cpu}(n_F) = (d_R + d_S) \cdot n_F \cdot \frac{|(R \bowtie S)_{sky}|}{B_2}
\]

The CPU cost for \(LESS\) [4] processing \(|R|\) records is:

\[
LESS_{cpu}(R) = k \cdot d_R \cdot ||R||,
\]

where \(k\) is a small constant.

The CPU cost for \(NLJ\) is \(|R||S|\) and that for \(INLJ\) is \(\alpha \cdot \beta \cdot ||R||\).

C. Motivation

From the above discussion, we can observe that the structure for the query evaluation has the general template of a join operation followed by a filter operation followed by the skyline evaluation. The steps towards optimizing the skyline query can follow two directions: (1) to further reduce the unnecessary input to the skyline operator, and (2) to design a pipeline query processing algorithm that overlaps the computation of the skyline and the join operators. The first direction aims to reduce the query cost by applying filtering techniques at different processing stages, and the second direction can be followed by studying the interactions between the join operator and the skyline operator, and allowing overlap between the computation of these two operations.

D. \(SMJ\)-Based Skyline Methods

In this subsection, we first propose two algorithms based on \(SMJ\). The first combines the \(LESS\) filtering step with the join processing, so that the join processing (with filtering) can overlap with the external sorting. We also propose a Partition-Filtering method, that partitions the smaller relation \(R\) into skyline tuples \(R_{sky}\) and non-skyline tuples \(R_{nonsky}\), and only needs to identify the skyline tuples in the joined tuples between \(R_{nonsky} \bowtie S\).

Algorithm 1 Early-Filtering Algorithm

1: Input: Relation \(R, S\), Join Attribute \(a^R, a^S\)
2: Output: Skyline_{RMS}
3: JoinResults = ∅
4: FilterWindow = ∅
5: while \(SMJ(R \bowtie S)\) Not Finished do
6:   Visit tuple \(r \in R, s \in S\)
7:   if \(a^R(r) >= a^S(s)\) then
8:     if \(r \bowtie s\) is not dominated by \(FilterWindow\) then
9:       JoinResults = JoinResults \cup r \bowtie s
10:      if \(r \bowtie s\) dominates any tuple \(t\) in \(FilterWindow\) then
11:         Remove \(t\) in \(FilterWindow\)
12:     end if
13:   else
14:     Replace worst-entropy tuple \(t\) in \(FilterWindow\)
15:   end if
16: end while
17: end if
18: if \(SizeOf(JoinResults) >= (B_1 - \gamma) / 2\) pages then
19:   Pipeline to \(LESS\)-Sort(JoinResults)
20: JoinResults = ∅
21: end if
22: end if
23: Visit next \(r, s\)
24: end if
25: end while
26: Return Skyline_{RMS} = LESS1(JoinResults)

1) Early-Filtering (EF): The early-filtering method prunes a join tuple as early as possible when it is produced, which saves the cost of writing join results to a temporary file. If the current join tuple is dominated by any tuple in the filtering window, then it is removed. The pseudo-code for \(EF\) is shown in Algorithm 1. It uses \(SMJ\) and consists of (a) a join operator that includes filtering (lines 6-17), and (b) a skyline operator using \(LESS\) that includes external sorting (lines 18-20 for pass zero and line 26 for the remaining passes). Since the join/filtering works in a pipeline, with the first round of external sorting, only half of the memory ((\(B_1 - \gamma)/2\)) can be used. \(LESS\)-Sort is the process for pass zero of the sorting, and \(LESS1\) for the remaining processing of the \(LESS\) algorithm.

This algorithm is blocking, i.e. the skyline operator \((SFS)\) starts evaluation only after the entire join processing (and the sorting) finishes.

The I/O cost of (a) is:

\[
C_1^{EF} = XSort_{10}(R) + XSort_{10}(S)
+ SMJ_{10}(R,S)
\]

(1)

and the CPU cost of (a) is:

\[
T_1^{EF} = XSort_{cpu}(R) + XSort_{cpu}(S)
+ Merge_{cpu}(R) + Merge_{cpu}(S)
+ \alpha \cdot \beta \cdot ||R|| \cdot n_F^{(1)} \cdot (d_R + d_S)
\]

(2)
Algorithm 2 Partitioning-Filtering Algorithm

1: Input: Relation \( R, S \), Join Attribute \( a^R_i, a^S_j \)
2: Output: Skyline_{RMS}
3: JoinResults = \emptyset
4: Skyline_{RMS} = \emptyset
5: Compute skyline Skyline_R for \( R \)
6: Remove dominated tuples in each group of \( S \) w.r.t \( a^S_j \)
7: JoinResultsSky1 = SMJ(Skyline_R \times S), Fill best-entropy tuples in FilterWindow while joining
8: while SMJ((R − Skyline_R) \times S) Not Finish do
9: Visit tuple \( r \in (R − Skyline_R), s \in S \)
10: if \( a^R_i(r) = a^S_j(s) \) then
11: Compare \( r \times s \) with tuples in FilterWindow
12: if \( r \times s \) is not dominated then
13: JoinResults = JoinResults \cup r \times s
14: if Sizeof(JoinResults) = (\( B_1 - \gamma - \eta \))/2 pages then
15: Compare tuples \( t \in \) JoinResults with tuples in JoinResultsSky1 − FilterWindow
16: if \( t \) is dominated then
17: JoinResults = JoinResults − \( t \)
18: Pipeline to LESS-Sort(JoinResults)
19: JoinResults = \emptyset
20: end if
21: end if
22: Visit next \( r, s \)
23: end if
24: end while
25: Return Skyline_{RMS} = LESS1(JoinResults)

The I/O cost of (b) is:

\[
C_{1PF}^F = XSort_{TO}(R) + XSort_{TO}(S) + SMJ_{TO}(Sky_r \times S) + |Sky_r \times S| + XSort_{TO}(R - Sky_r) + SMJ_{TO}(R - Sky_r) - SMJ_{TO}(Sky_r \times S) - |Sky_r| - |Sky_r| (5)
\]

and the CPU cost of (b) is:

\[
T_{1PF}^F = XSort_{cpu}(R) + XSort_{cpu}(S) + Merge_{cpu}(S) + \alpha \cdot \beta \cdot |Sky_r| + XSort_{cpu}(R - Sky_r) + Merge_{cpu}(R - Sky_r) + \alpha \cdot \beta \cdot |R_{nonSky}| \cdot |Sky_r \times S| + (d_R + d_S) (6)
\]

The filtering window is filled with the best dominating tuples during the joining process, using the same criteria as LESS for determining these best tuples. Then, SMJ is applied between \( R_{nonSky} \) and \( S \), and the join tuples are compared with the already generated skyline tuples. Those tuples that are not pruned are then sent to the LESS module for skyline computation. The pseudo-code of this method is shown in Algorithm 2, which consists of (a) a join operator that includes a skyline partition join operation (lines 5-7), a non-skyline partition join operation including filtering (lines 8-18), and (b) a skyline operator that includes external sorting (line 19 for pass zero and line 26 for the remaining passes) and SFS (also in line 26).

Here we assume \( \eta \) is the number of pages allocated to store tuples in JoinResultsSky1 partially or fully depending on the size of JoinResultsSky1 (if the size of JoinResultsSky1 is less than \( \eta \) pages, then we use the actually needed storage space, otherwise we use \( \eta \) pages). The number of pages \( \eta \) used to store skyline tuples is defined as: \( \eta = \min(\eta, |JoinResultsSky1|) \).

The I/O cost of (a) is:

\[
C_{2PF}^F = \frac{XSort_{TO}(n^R_F) + SFS_{TO}(n^S_F) \cdot (R \times S)_{sky}}{B_s} (3)
\]

and the CPU cost of (b) is:

\[
T_{2PF}^F = XSort_{cpu}(n^R_F) + Merge_{cpu}(n^R_F) + SFS_{cpu}(n^S_F) (4)
\]

The total time cost is \( (C_{1PF}^F + C_{2PF}^F) \cdot w + T_{1PF}^F + T_{2PF}^F \).

2) Partition-Filtering (PF): This method partitions the smaller relation \( R \) into skyline tuples \( Sky_r \) and non-skyline tuples \( R_{nonSky} \) (the cost of finding the skyline in a smaller table is cheaper). For relation \( S \) during the joining process, the skyline in each join group (tuples that have the same join attribute value) is computed. In general, this is a relatively cheap operation since the size of each group is small relative to the size of the relation and the size of memory. Applying SMJ between \( Sky_r \) and the group skylines of \( S \) produces the resulting skyline. This technique also has one big benefit when performing the join: no further dominance comparisons are required with other tuples.

E. NLJ-based Skyline Methods

Although the blocking skyline evaluation (described above) reduces the number of unnecessary tuples sent to the skyline operator and overlaps the join processing (and filtering) with external sorting, it can only return skyline answers after all the input tuples finish the join and the sort processing.
Algorithm 3 Block Nested Loop Pipelining Algorithm

1: Input: Relation R, S. Join Attribute aR i, aS j
2: Output: Skyline_{RMS}
3: JoinResults = ∅
4: Skyline_{RMS} = ∅
5: Sort R w.r.t. entropy
6: Sort S w.r.t. entropy
7: for each block of B1/3 pages of R do
8:     for each page of S do
9:         for all in-memory tuples r ∈ R-block, s ∈ S-page
10:            if aR i (r) == aS j (s) then
11:                JoinResults = Merge(JoinResults, r × s)
12:            if Sizeof(JoinResults) == (B1-1)/3 pages
13:                Pipeline to SFS(JoinResults)
14:         end if
15:     end for
16: end for
17: Return Skyline_{RMS} = SFS(JoinResults)

Similar to the discussion of buffer usage presented in the Early-Filtering section, for B1 buffer pages, one page is used to read data from S, while B1/3 pages are used for reading blocks of data from R for joining. The remaining buffer is split into two parts: one part for storing joined results and one part for SFS to process the skyline evaluation.

The I/O cost of (a) is:

$$C_1^{BNLP} = XSort_{IO}(R) + XSort_{IO}(S) + 3 \cdot BN LJ_{IO}(R, S).$$

The CPU cost of (a) is:

$$T_1^{BNLP} = XSort_{cpu}(R) + XSort_{cpu}(S) + \|R\| \cdot \|S\|.$$  

The I/O cost of (b) is:

$$C_2^{BNLP} = SFS_{IO}(n^{(p)}_1, (R \times S)_{sky}) + \|((R \times S)_{sky})\| \cdot (d_R + d_S)$$

and the CPU cost of (b) is:

$$T_2^{BNLP} = SFS_{cpu}(n^{(p)}_1) \cdot (d_R + d_S)$$

2) Index Nested Loop Pipelining (INLP) Skylines: The traditional skyline method processes tuples one by one, consuming CPU cycles while reading them from disk. This method combines the processing of skyline values with the join during the join scanning process, and the result is a sequence of tuples with skyline values.

We remove those tuples which are dominated by other tuples in the group. Then, we pipeline the join results to SFS directly instead of performing filtering or external sorting.

The pseudo-code for this method is shown in Algorithm 4, which consists of (a) sorting R on entropy value (line 5), (b) a join operator that generates joined tuples (lines 6-17), and (c) a skyline operator for skyline computing (SFS line 22).

Note that θ is a parameter that is the maximum number of pages needed to store the skyline of a group with the same join attribute value in S. In case the actual number of pages needed is greater than θ, then the tuples are processed in batches.

The I/O cost of (a) is:

$$C_1^{INLP} = XSort_{IO}(R) + IN LJ_{IO}(R, S).$$

The CPU cost of (a) is:

$$T_1^{INLP} = XSort_{cpu}(R) + Merge_{cpu}(R) + \alpha \cdot \beta \cdot \|R\| \cdot (1 + \rho).$$

The I/O cost of (b) is:

$$C_2^{INLP} = SFS_{IO}(n^{(p)}_1, (R \times S)_{sky}) + \|((R \times S)_{sky})\| \cdot (d_R + d_S)$$

and the CPU cost of (b) is:

$$T_2^{INLP} = SFS_{cpu}(n^{(p)}_1).$$
Algorithm 4 Index Nested Loop Pipelining Algorithm

1: Input: Relation \( R \), index of \( S \), Join Attribute \( a^R_i, a^S_j \)
2: Output: Skyline_{R,M}
3: JoinResults = \( \emptyset \)
4: Skyline_{R,M} = \( \emptyset \)
5: Sort \( R \) w.r.t. entropy
6: for each tuple \( r \in R \) do
7: \( Temp = \emptyset \)
8: for each tuple \( s \in S \) satisfies \( a^R_i(r) = a^S_j(s) \) do
9: Compare \( s \) with tuples in \( Temp \)
10: if \( s \) is not dominated then
11: \( Temp = Temp \cup s \)
12: end if
13: if \( s \) dominates a tuple \( t \) in \( Temp \) then
14: Remove \( t \) in \( Temp \)
15: end if
16: end for
17: JoinResults = Merge(JoinResults, \( r \bowtie Temp \))
18: if Sizeof(JoinResults) \( = (B_1 - \theta)/2 \) pages then
19: Pipeline to SFS(JoinResults)
20: end if
21: end for
22: Return Skyline_{R,M} = SFS(JoinResults)

The total cost is \( (C_1^{INLP} + C_2^{INLP}) \cdot w + T_1^{INLP} + T_2^{INLP} \).

For the above two algorithms, we use SFS to output skyline tuples progressively. Although SFS requires that the input tuples to be sorted by their entropy values, we prove that BNLP and INLP (Algorithm 3 and Algorithm 4) can produce the skyline progressively, even when the joined tuples produced in these algorithms may not satisfy this requirement.

**Theorem 1:** Algorithm 3 (BNLP) and Algorithm 4 (INLP) produce skyline results progressively.

**Proof:** (1) In Algorithm 3, suppose the tuples in \( R,S \) sorted by entropy values are \( r_1, \ldots, r_{|R|}, s_1, \ldots, s_{|S|} \) respectively. For any tuple \( r_i \), since \( Ent(r_i) \leq Ent(r'_i) \) \((i < i')\), \( r_i \) cannot be dominated by \( r_{i'} \). Similarly, any tuple \( s_j \) cannot be dominated by tuple \( s_{j'} \) \((j < j')\). For any \( r_i, r_{i'} \) \((i < i')\) and \( s_j, s_{j'} \) \((j < j')\), suppose \( Ent(r_i) < Ent(r_{i'}) \). If \( r_i \) joins \( s_j \) and \( r_{i'} \) joins \( s_{j'} \), \( r_i \bowtie s_j \) cannot be dominated by \( r_{i'} \bowtie s_{j'} \). If \( r_i \) joins \( s_{j'} \) and \( r_{i'} \) joins \( s_j \), \( r_i \bowtie s_{j'} \) cannot be dominated by \( r_{i'} \bowtie s_j \) either although \( Ent(r_i \bowtie s_{j'}) \) might be larger than \( Ent(r_{i'} \bowtie s_j) \). If \( Ent(r_i) = Ent(r_{i'}) \) but their skyline attribute values are not equal, the joined tuples satisfy NDO-property. If \( Ent(r_i) = Ent(r_{i'}) \) and both have the same skyline attribute values, the join order of \( r_i \bowtie s_j \) and \( r_{i'} \bowtie s_{j'} \) follows non-dominated ordering. If \( r_i \bowtie s_{j'} \) is generated before \( r_{i'} \bowtie s_j \), the Merge() function (line 11) will switch their order in the join results to keep this non-dominated ordering property. Therefore skyline objects are found progressively. (2) Similarly, we can also prove this same property for Algorithm 4.

IV. EVALUATING SKYLINE QUERIES

In order to allow the query optimizer to produce efficient plans for evaluating skyline queries with joins, it is important to estimate the size of the skyline over any joined table. For example, the query optimizer can select different skyline algorithms according to the cardinality of the skyline results. If the join size is large (say over 1 million joined tuples), while the skyline cardinality is limited (say only several hundred), then a good skyline query plan would return to users the skyline progressively during the join processing, instead of waiting for the end of the join operation.

A. Skyline Cardinality Estimate When the Join Size is Known

We first assume that the cardinality of the join table is already determined (or estimated using existing methods).

1) Problem with a Naive Solution: One might think of estimating the number of skyline objects in a join table by first calculating the size of the join table, and then applying some classic skyline size estimation method, e.g., in [16]. Unfortunately, this naive solution does not produce good results. Recall that the classic skyline size estimation method requires that all attributes be independent, and that all values of the same attribute be distinct. However, the joined table does not meet this requirement since one tuple \( r \) with join attribute value \( val \) in table \( R \) may correspond to several tuples with the same join attribute value \( val \) in \( S \), so that the attribute values of the \( r \) tuple are repeated multiple times in the joined table. In addition, all join partners of one tuple have identical (i.e., not independent) attribute values. Hence, we need to estimate the number of skyline objects directly from the original tables. In [17], the authors study the cardinality estimation of join \( T_1 \bowtie T_2 \) in a very special case in which only attributes in \( T_1 \) participate in skyline evaluation. However, a more accurate estimate will result by considering the general case where attributes in both \( T_1 \) and \( T_2 \) participate in the skyline evaluation.

2) The Estimation Model: Let \( R \) and \( S \) denote two tables consisting of \( d_R \) and \( d_S \) attributes for skyline evaluation, respectively, where each has one join attribute. We make two basic assumptions: First, every data attribute of \( R \) or \( S \) is drawn randomly from some probability distribution and is distinct. Second, all attributes of \( R \) or \( S \) are independent (note that this is only for the input tables but not for the join table).

The join attributes of \( R \) and \( S \) both range on some discrete set \( \Omega = \{\omega_1, \ldots, \omega_t\} \). Note that \( R \bowtie S \) consists of \( d_R + d_S \) attributes and two join attributes. Our goal is to estimate the number of skyline records in \( R \bowtie S \) solely on the \( d_R + d_S \) attributes.

Theorem 2: Suppose \( a = (a_1, a_2) \) is a tuple participating in skyline evaluation and \( \omega = (\omega_1, \omega_2) \) is a skyline attribute. The naive solution produces an estimation \( \hat{T}(a, \omega) \) but the true total number of skyline tuples, \( T(a, \omega) \), is generally lower. To rectify this, we define a new attribute \( D(a, \omega) \) that is equal to \( T(a, \omega) \) if \( \hat{T}(a, \omega) \) is equal to \( T(a, \omega) \), but otherwise, it is equal to \( \hat{T}(a, \omega) \). Let \( D \) be the set of tuples participating in skyline evaluation.

\[ T(a, \omega) = \sum_{a \in D} D(a, \omega) \]

for all \( a \).
3) The Size of the Join Table: Since every record in \( R \) joins with each record in \( S \) with the same value on the join attribute, \( R \times S = \bigcup_i R_i \times S_i \), and \( |R \times S| = \sum_i r_i \cdot s_i \).

4) The Expected Number of Skyline Objects in the Join Table: To estimate the number of skyline objects, we fix an arbitrary record \( z = x \times y \in R_i \times S_i \) where \( x = (x_1, \ldots, x_d) \in R_i \), \( y = (y_1, \ldots, y_d) \in S_i \). Next, we calculate the probability that \( z \) is a skyline tuple (i.e., none of the records in \( R \times S \) dominates \( z \)). Note that for \( z \in R_i \times S_i \), to be a skyline object, the following lemma holds.

**Lemma 2:** A joined tuple \( z = x \times y \,(x \in R_i, y \in S_i) \) is a skyline if (1) none of the records in \( R_i \) dominates \( x \) and none of the records in \( S_i \) dominates \( y \), and (2) for any \( j \neq i \), none of the records in \( R_j \) dominates \( x \), or none of the records in \( S_j \) dominates \( y \).

**Proof:** If \( x \) is found to be dominated by \( x' \in R_i \), then \( x \times y \) is always dominated by \( x' \times y \), which contradicts the fact that \( z = x \times y \) is a skyline, so (1) is proved. Similarly we can prove (2).

Based on Lemma 2, integrating over \( x \) and \( y \), we get  
\[
\Pr[z \text{ is skyline}] = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \prod_{j=1, j \neq i}^{\infty} \{\Pr[-(R_j > x) \text{ or } -(S_j > y)]\} \cdot \Pr[-(R_i > x) \text{ and } -(S_i > y)] \prod_{k=1}^{d_R} dF(x_k) \prod_{k=1}^{d_S} dF(y_k)
\]

Without loss of generality, assuming the components of each vector \((x_1, \ldots, x_d)\) have continuous distribution functions \( F(x_i)(i = 1, \ldots, d) \). Unfortunately, it is difficult to solve the above integration directly. Instead, we deduce robust upper and lower bounds. Let \( p(m, d) \) be the probability that a random \( d\)-dimensional record is a skyline member among \( m \) random records. Note that \( p(m, d) \approx \frac{\log(m)^{d+1}}{m(d+1)} \), according to [18].

5) Upper Bound: To show the upper bound, we need the following lemma which can be proved using the majorization technique.

**Lemma 3:** \( \forall p, q, k : 0 < p < 1, 0 < q < 1, k \geq 1, (1 - p)^k + (1 - q)^k - (1 - pq)^k \leq (1 - (1 - p), 1 - q))^k \).

**Proof:** Note that \( 1 - pq \geq \max\{1 - p, 1 - q\} \) since \( 0 < p, q < 1 \). Moreover, \( (1 - p) + (1 - q) = (1 - p) \cdot (1 - q) + (1 - pq) \), hence \( (1 - p, 1 - q) \preceq ((1 - p)(1 - q), 1 - pq) \). The lemma follows from the fact that the function \( f(x) = \sum x_i^k \) is Schur-convex. For more detail about majorization and Schur-convex, interested readers can check [19].

By inclusion-exclusion and Lemma 3, we get \( \forall j \neq i, \Pr[-(R_j > x) \text{ or } -(S_j > y)] \).

\[
\Pr[-(R_i > x) \text{ and } -(S_i > y)] \prod_{k=1}^{d_R} dF(x_k) \prod_{k=1}^{d_S} dF(y_k)
\]

Let \( a = \sum_{j=1, j \neq i}^{\infty} \min \{r_j, s_j\} \) and \( a_{-i} = \sum_{j=1, j \neq i}^{\infty} \min \{r_j, s_j\} \). By Equation 17, \( \Pr[z \text{ is skyline}] \)

\[
\leq \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \prod_{j=1, j \neq i}^{\infty} \{\Pr[-(R_j > x) \text{ or } -(S_j > y)]\} \cdot \Pr[-(R_i > x) \text{ and } -(S_i > y)] \prod_{k=1}^{d_R} dF(x_k) \prod_{k=1}^{d_S} dF(y_k)
\]

Finally, since there are \( |R \times S| \) records in the join table, the expected number of skyline objects in \( R \times S \) is upper bounded by \( |R \times S| \cdot p(a - 1, d_R + d_S) \).

6) Lower Bound: Since attributes of \( R, S \) are independent, we have  
\[
\Pr[-(R_i > x) \text{ and } -(S_i > y)] = \Pr[-(R_i > x)] \cdot \Pr[-(S_i > y)]
\]

Meanwhile, since  
\[
\Pr[-(R_i > x)] > \Pr[-(R_i > x) \text{ and } -(S_i > y)]
\]

We have  
\[
\Pr[-(R_i > x) \text{ and } -(S_i > y)] = \Pr[-(R_i > x)] \cdot \Pr[-(S_i > y)] - \Pr[-(R_i > x) \text{ and } -(S_i > y)]
\]

Thus, by Equation 17, \( \Pr[z \text{ is skyline}] \)

\[
\geq \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \prod_{j=1, j \neq i}^{\infty} \{\Pr[-(R_j > x) \text{ or } -(S_j > y)]\} \cdot \Pr[-(R_i > x)] \prod_{k=1}^{d_R} dF(x_k) \prod_{k=1}^{d_S} dF(y_k)
\]

Consequently, the number of skyline objects in \( R \times S \) is lower bounded by \( p(|R|, d_R) \sum_{i=1}^{\infty} r_i \cdot s_i \cdot p(s_i, d_S) \). In summary, the number of tuples in the joined skylines is between \( p(|R|, d_R) \sum_{i=1}^{\infty} r_i \cdot s_i \cdot p(s_i, d_S) \) and \( |R \times S| \cdot p(a - 1, d_R + d_S) \).
B. Skyline Cardinality Estimate When the Join Size is Unknown

If the join size ($\sum_i r_i \cdot s_i$) is not known and still needs to be estimated, some existing methods can be used to estimate this size. For example, the recently developed "End-biased" sampling method [20] provides more accurate estimates than random samples with the same sample size. It picks samples based on two criteria: preferentially sampling values that repeat more often, and correlating sampling decisions in different tables to help find infrequent values that occur in both. The theoretical analysis in this paper shows that when using the same number of elements in the sample, end biased samples always gives more accurate (lower variance) results than random samples.

When the join size of two tables with $n$ tuples is at least $B$ (typically values for $B$ are $n^2$ or $n \log n$), we can apply the method developed by Alon et al. [21] by using a uniform random sample of size $cn^2/B$ tuples, which can ensure accurate estimates with high probability, where $c > 3$ is a constant that depends on the desired accuracy and confidence.

With the estimated join size, we can plug-in the same cardinality analysis in the previous section and obtain the corresponding lower bound and upper bound.

C. An Analytical Evaluation of the Proposed Methods

Given the estimates of the skyline cardinality as well as other factors, including the size of the $R$ and $S$ relations, the size of $R \not\prec S$, and the selection of the join algorithms, we now discuss which query evaluation method is likely to be preferable under different circumstances. As the cost of skyline computation in existing work is measured under the independent distribution assumption, our cost measurement also uses this assumption.

Amongst the algorithms for progressive skyline evaluation, the index nested loop pipelining method (INLP) is a better choice, since each cost in formula (13), (14), (15), and (16) is always smaller than the cost in formula (9), (10), (11), and (12) respectively. For the early filtering (EF) method, since $|S| \geq |R|$ and $c$ is usually a small constant, we can show from formulas (1) and (2), and formulas (13) and (14) that $C_{1}^{EF} \leq C_{1}^{INLP}$, and $T_{1}^{EF} \leq T_{1}^{INLP}$. Consider formula (3) and (4) for part (b) in Algorithm 1 and formula (15) and (16) in Algorithm 4, since $n_F^{(1)} \approx n_F^{(4)}$ (under the independent distribution assumption), we have $C_{2}^{EF} > C_{2}^{INLP}$ and $T_{2}^{EF} \geq T_{2}^{INLP}$. So the INLP method is a better plan than EF.

We also compare partition filtering method (PF) with INLP. After summing $C_{1}^{PF}$(5) and $C_{2}^{PF}$(7), since $|S| \geq |R|$ and $c$ is a small matching constant, it is easy to derive $C_{1}^{PF} + C_{2}^{PF} > C_{1}^{INLP} + C_{2}^{INLP}$. Similarly, we sum $T_{1}^{PF}$ in (6) and $C_{2}^{PF}$ in (8), and have $T_{1}^{PF} + T_{2}^{PF} > T_{1}^{INLP} + T_{2}^{INLP}$. So INLP is a better plan than PF method. Both the EF method and the PF method behave better than the BNLP method since BNLP includes an expensive join step and does not do any intermediate filtering, which can be illustrated by comparing $C_{1}^{BNLP}$(9), $T_{1}^{BNLP}$(10), $C_{2}^{BNLP}$(11) and $C_{2}^{BNLP}$(12) with the corresponding partner in EF and PF.

We further study the blocking skyline evaluation methods, EF and PF, by examining their sorting, joining, filtering (which includes building the filter window and doing the actual filtering), and LESS processing steps. In terms of sorting, they all have to sort $R$ and $S$ first (although PF splits $R$ into a skyline partition and a non-skyline partition), so the I/O and CPU costs are almost the same for both methods. In terms of the join costs, each method scans $R$ and $S$ sequentially to go through all the joined tuples (though again PF does this in two steps). For the filtering process, EF does the window building and filtering during the joining, and the filtering capability increases when more tuples are joined. PF builds a good filter window through the first round of the join. The final skyline objects $|R_{sky} \times S|$ (which is typically smaller than $|R \times S|$) are generated in this round (the percentage of $|R_{sky} \times S|$ contained in the complete set of skyline points depends on the distribution). After the joining/filtering step is done, the LESS process starts (the initial sorting phase can go in parallel with joining/filtering as we presented earlier with no extra filtering required for LESS as it is already done during the joining). As PF builds a better dominating filter window and part of the joined skyline is already evaluated during the joining process, PF is expected to be more competitive, especially when the resulting skyline has large cardinality (for example, if $R$ has an anti-correlated distribution).

In summary, when the skyline attributes in both input tables follow independent distributions, the INLP method is expected to be the best choice, PF is expected to be more competitive than EF when join tables are of different sizes. BNLP is expected to be fourth in terms of total cost. However, if there is no index support for the larger relation $S$ and the user wants the skyline returned progressively, then BNLP can still be a good choice.

When any input table follows an anti-correlated distribution, the skyline cardinality grows and the skyline computation cost for each method increases as well. However, until now there has been no general way to measure the cost in this case. In the experimental section, we observe that INLP is still competitive due to its use of an index and progressive processing and that the PF method may be the best algorithmic choice if progressive output is not required.

V. Experimental Evaluation

All the methods presented in this section are implemented in C++. These methods include (1) two baseline algorithms: a sort-merge join based skyline method (SMJ) where joined tuples are first generated by SMJ and then processed by LESS, and a block nested loop based skyline method (BNL) where joined tuples are first generated by BNL and then processed by LESS, (2) the early-filtering (EF) method, (3) the partitioning filtering (PF) method, (4) the block nested-loop pipelining skyline (BNL) method; and (5) the index nested-loop pipelining skyline (INLP) method.

All experiments were carried out on an Intel Pentium 4 2.2 GHz duo CPU machine with 2GB main memory and a 160 GB hard disk, running Microsoft Windows Vista. We implemented
a buffer pool with the CLOCK replacement policy, and used 4KB sized pages. The number of default buffer pages for skyline processing is 200. The size of the LESS filtering window is 5 pages, the number of records in a block of BNL join is 500, and the size of each attribute is 4 bytes.

A. Synthetic Data Test

For this test, we generate two datasets, denoted as $R$ and $S$, each with one (default) join attribute and some skyline evaluating attributes. We first use the data generator in [1] to generate 10,000, 3-d relation ($R$) and a 100,000 3-d relation ($S$) and test the skyline queries on 9 combinations of independent ($I$), correlated ($C$) and anti-correlated ($A$) distributions for these relations. We assume that a B+-tree index is prebuilt on the join attribute of $S$.

1) Input Tables with Different Data Distribution: For each pair of ($R$, $S$), where the size and dimensionality of $R$ and $S$ are chosen as above, we change the data distributions of skyline evaluation attributes in $R$ and $S$, to produce nine combinations in total: ($R_C$, $S_C$), ($R_C$, $S_I$), ($R_I$, $S_C$), ($R_I$, $S_I$), ($R_A$, $S_I$), ($R_I$, $S_A$), ($R_A$, $S_C$), ($R_C$, $S_A$), and ($R_A$, $S_A$). For example, ($R_C$, $S_I$) refers to the configuration of $R$ with the correlated distribution and $S$ with the independent distribution. The join attribute values follow the independent distribution and are randomly sampled from [0,10k]. The cardinality of the join results in this experiment is 104,271. The run time of the skyline evaluation methods are listed in Table II. These results show that INLP achieves better performance than the other methods. Additionally, EF becomes more efficient when more skyline objects appear in $R$ (especially if $R$ is anti-correlated).

2) Scalability Test: The scalability of the methods presented here is evaluated by choosing two independent dataset distributions, $R_I$ and $S_I$, each with dimensionality of 3. The cardinality of $S$ is 10 times that of $R$, and we vary the cardinality of $R$ from 10k to 100k. Values of the join attribute in $R$ and $S$ follow independent distributions and are generated by randomly sampling from the integer domains [0,10k), [0, 20k), [0,40k), [0,80k) and [0,100k) with increasing relation sizes. The performance results are shown in Figure 2. These results show that all the algorithms scale well with increasing input table sizes, except BNL and BNP. These methods indicate similar performance initially when the size of the input tables is small. From the point when the size of the $R$ relation is 40,000 and the size of the $S$ relation is 400,000 and beyond, INLP runs faster than all competing methods, PF ranks second, and EF is third.

The setup of the second scalability test is similar to the previous test except the input tables are $R_A$ and $S_I$ (the anti-correlated distribution against the independent distribution). The results for this test are shown in Figure 3. This figure shows that with increasing input table sizes, every method in this experiment takes more time than the previous test (Figure 2). Now PF is the fastest method due to its performance advantage on anti-correlated data. The performance of INLP is close to PF, and SMJ is slightly faster than EF.

The third scalability test has the same configuration as before (an anti-correlated distribution joined with an independent distribution), except that the resulting output size of the join is now 10 times larger. This is accomplished by reducing the cardinality of the join attribute domain by 1/10. Figure 4 shows the results for this experiment. Here the performance trend is similar to that of Figure 2, with the difference that BNL is slightly better than BNL. This experiment also shows that these algorithms scale well with an increasing number of join results.

In the fourth scalability experiment (with results in Figure 5), we change the data distributions as $(R_I, S_A)$, and keep the same configuration as in the previous scalability test. We
observe from Figure 5 that each method scales well with the increasing size of the input tables. However, the overall query evaluation time is generally higher than the previous experiment (Figure 4). \textit{INLP} finishes first, and \textit{PF} and \textit{SMJ} finish second and third respectively.

To evaluate the effect of varying the dataset dimensionality, we use input datasets $R_I$ and $S_I$ with cardinality 10$k$ and 100$k$, respectively. These two datasets have join attribute values that are chosen independently from $[0, 10k]$. We vary the dimensionality in the joined table from 3 to 7. These results are shown in Figure 6. This figure shows that \textit{INLP} outperforms all other methods. \textit{PF} takes less time than the remaining methods, and \textit{EF} ranks third.

Next we examine the I/O cost, which is measured as the number of buffer pool misses. The setup of this experiment is identical to the first scalability test (Figure 2). These results are shown in Figure 7, which indicates that \textit{INLP} incurs the least number of disk I/Os, while \textit{PF} and \textit{EF} incur fewer I/Os than the remaining methods. (As a consistency check, we have also evaluated formulas (1) to (14) with similar parameters as the experiment shown in Figure 7, and obtain similar results.)

To compare the efficiency of the progressive evaluation methods, we measure the time to return the first-1, first-3, first-5, first-10, first-20, first-30, first-50, first-100, first-200 and first-300 skyline objects on $R \bowtie S$, where $|R| = 100k$, $|S| = 1000k$, and both datasets are distributed independently. From Figure 8, we observe that \textit{INLP} is the quickest in returning the answers, followed by \textit{BNLP}. The other methods are far behind.

We also investigate the accuracy of our skyline cardinality estimation technique by comparing the actual number of skyline objects with the proposed lower and upper bound. These results are shown in Figure 9. The setup for this test is similar to the first scalability test (Figure 2) except that the dimensionality of each input table is 4. As shown in Figure 9, the actual skyline cardinality is between our lower bound and upper bound. In particular, our lower bound is close to the actual size of the skyline objects even when the input table sizes increase to the large extent (e.g. $|R| = 100k$, $|S| = 1000k$).

B. Real Data Test

We have also evaluated these methods using two real datasets from the commonly used NBA players dataset [22]. These datasets include (1) 6,982 team salary records, with one dimension of team salary, and one dimension of the team ID and (2) 11,150 NBA players’ statistics records, with one dimension of team ID, and five dimensions of PTS, REB, AST, BLK, and players’ Salary. The join attribute is the team ID. The results for this test are shown in Table III. As can be observed from this table, \textit{INLP} outperforms the other methods, and is closely followed by \textit{PF} and \textit{EF}.
In this paper, we have proposed several methods to evaluate the skyline operator in the presence of equijoin operations, and developed algorithms to incorporate the state-of-the-art join methods into skyline computation. We analyzed the cost of these methods and present a novel method to estimate the skyline cardinality over the joined table. Various evaluation methods for skyline queries based on the estimation method and cost analysis are also presented.

We perform experimental evaluations on both synthetic datasets and the NBA dataset and conclude that: (1) the INLP method, which is an index-based nested loop method, works better when the skyline attributes in both the input tables have independent distributions. It is also the fastest progressive skyline method among the proposed methods; (2) The filtering-based methods PF and EF methods are better than the BNLP method under the independent data distributions; (3) When the inputs do not have pre-built indices, and the users wish to progressively retrieve skyline results, then the BNLP method is a viable alternative. Alternatively, we can consider building an index on the fly and using the INLP method. These results demonstrate the cost analysis of the proposed evaluation methods, and validates the effectiveness of our framework.

### VI. Conclusions

In this paper, we have proposed several methods to evaluate the skyline operator in the presence of equijoin operations, and developed algorithms to incorporate the state-of-the-art join methods into skyline computation. We analyzed the cost of these methods and present a novel method to estimate the skyline cardinality over the joined table. Various evaluation methods for skyline queries based on the estimation method and cost analysis are also presented.

We perform experimental evaluations on both synthetic datasets and the NBA dataset and conclude that: (1) the INLP method, which is an index-based nested loop method, works better when the skyline attributes in both the input tables have independent distributions. It is also the fastest progressive skyline method among the proposed methods; (2) The filtering-based methods PF and EF methods are better than the BNLP method under the independent data distributions; (3) When the inputs do not have pre-built indices, and the users wish to progressively retrieve skyline results, then the BNLP method is a viable alternative. Alternatively, we can consider building an index on the fly and using the INLP method. These results demonstrate the cost analysis of the proposed evaluation methods, and validates the effectiveness of our framework.

### VII. Acknowledgments

This research was supported in part by the National Science Foundation under grant IIS-0929988.

### REFERENCES


