PrefJoin: An Efficient Preference-aware Join Operator

Mohamed E. Khalefa [1], Mohamed F. Mokbel [2], Justin J. Levandoski [3]

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA

[1]`khalefa@cs.umn.edu`, [2]`mokbel@cs.umn.edu`, [3]`justin@cs.umn.edu`

***Abstract*—Preference queries are essential to a wide spectrum of applications including multi-criteria decision-making tools and *personalized* databases. Unfortunately, most of the evaluation techniques for preference queries assume that the set of preferred attributes are stored in only one relation, waiving on a wide set of queries that include preference computations over multiple relations. This paper presents *PrefJoin*, an efficient preference-aware join query operator, designed specifically to deal with preference queries over multiple relations. *PrefJoin* consists of four main phases: *Local Pruning*, *Data Preparation*, *Joining*, and *Refining* that filter out, from each input relation, those tuples that are guaranteed not to be in the final preference set, associate meta data with each non-filtered tuple that will be used to optimize the execution of the next phases, produce a subset of join result that are relevant for the given preference function, and refine these tuples respectively. An interesting characteristic of *PrefJoin* is that it tightly integrates preference computation with join hence we can early prune those tuples that are guaranteed not to be an answer, and hence it saves significant unnecessary computations cost. *PrefJoin* supports a variety of preference function including skyline, multi-objective and k-dominance preference queries. We show the correctness of *PrefJoin*. Experimental evaluation based on a real system implementation inside PostgreSQL shows that *PrefJoin* consistently achieves from one to three orders of magnitude performance gain over its competitors in various scenarios.**

I. Introduction

Preference queries are essential to a wide spectrum of applications including multi-criteria decision-making tools and personalized databases [15], [17]. Several preference functions have been proposed in the literature including *top-k* [7], *skylines* [3], *multi-objective* [2], *k-dominance* [5], *k-frequency* [6], and *ranked skylines* [18]. Given a set of multi-dimensional tuples, preference queries find a set of interesting tuples, i.e., tuples that are preferred to the user according to some preference function. An example preference query is *"find my best restaurants based on my preferences"* where user preferences for a restaurant can be minimal price and minimal distance.

Most of the research efforts for the preference query evaluation are designed to compute the preference set over a single relation (e.g., see [1], [3], [4], [7], [8], [10], [16], [18], [25], [34]). Unfortunately, such work can not be directly applied to a wide spectrum of preference applications and queries where the preferred attributes are stored in more than one relation.

This work is supported in part by the National Science Foundation under Grants IIS-0811998, IIS-0811935, CNS-0708604, IIS-0952977 (NSF CAREER) and by a Microsoft Research Gift

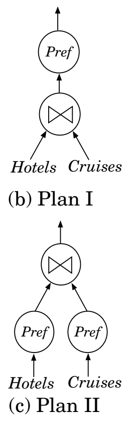
HId	*Location*	*P*	*R*	*B*
1	*Miami*	*4*	*3*	*1*
2	*Miami*	*4*	*2*	*3*
3	*Miami*	*3*	*3*	*3*
4	*Miami*	*4*	*4*	*3*
5	*Miami*	*4*	*3*	*5*
6	*Hawaii*	*2*	*2*	*2*
7	*Hawaii*	*5*	*3*	*1*
8	*Hawaii*	*4*	*1*	*4*
9	*Hawaii*	*5*	*5*	*6*
10	*Hawaii*	*5*	*4*	*7*
11	*Seattle*	*3*	*3*	*1*
12	*Seattle*	*3*	*1*	*4*
13	*Seattle*	*2*	*3*	*4*
14	*Seattle*	*4*	*4*	*3*
15	*Seattle*	*5*	*5*	*6*

Hotels

CId	*Location*	*P*	*R*	*D*
1	*Miami*	*4*	*3*	*3*
2	*Miami*	*5*	*2*	*3*
3	*Miami*	*5*	*3*	*1*
4	*Miami*	*4*	*4*	*6*
5	*Miami*	*4*	*3*	*5*
6	*Hawaii*	*5*	*3*	*1*
7	*Hawaii*	*3*	*2*	*2*
8	*Hawaii*	*4*	*1*	*4*
9	*Hawaii*	*5*	*2*	*8*
10	*Hawaii*	*7*	*5*	*6*
11	*Seattle*	*4*	*3*	*1*
12	*Seattle*	*5*	*1*	*4*
13	*Seattle*	*3*	*3*	*4*
14	*Seattle*	*4*	*4*	*5*
15	*Seattle*	*6*	*5*	*6*

Cruises

(a) Relations Hotels & Cruises



(b) Plan I

(c) Plan II

Fig. 1. Motivating Example

Consider, for example, a scenario where a user is looking for a vacancy destination. for simplicity assume that the user is only concerned about the hotel and the cruise. User preferences for the hotel are lower price, better rating, and closer to the beach and for the cruise are lower price, better rating, and shorter stay. Figure 1a gives information about hotels and cruises, stored in relations *Hotels* and *Cruises*, respectively. Assuming minimum rating is better, this user preference request can be represented by the following SQL query:

```
SELECT * from Hotels h, Cruises c
WHERE c.location = h.location
PREFERENCE h.price(min), h.rating(min),
h.beach_dist (min), c.price(min),
c.rating(min), c.days(min)
```

To answer this SQL query using existing single-table preference techniques, we first need to join the two input relations (i.e., *Hotels* and *Cruises*), and then apply the preference query algorithm on the joined relation based on the location attribute, i.e., `Pref` (*Hotels* $\bowtie_{location}$ *Cruises*). Figure 1b shows such a query plan. Unfortunately, such an approach is very inefficient as it completely isolates the preference functionality from the join operator. As a result, the join operation will produce too many tuples that have no chance of being preferred tuples. Another approach is to push the single-relation preference operation `Pref` before the join operator, as depicted in the query plan in Figure 1c. However, this is simply incorrect as the preference is not distributive over the join, i.e., Pref ($Hotels \bowtie_{location} Cruises$) $\neq$ Pref($Hotels$)$\bowtie_{location}$Pref($Cruises$). For example, in Figure 1, if the preference function is a skyline query, the hotel represented by tuple (2,$\underline{Miami}$,4,2,3) is dominated by hotel (6,$\underline{Hawaii}$,2,2,2), hence it would not proceed to the join operator. Similarly, cruise (3,$\underline{Miami}$,5,3,1) is dominated by cruise (11,$\underline{Seattle}$,4,3,1). However, the joined tuple (2,$\underline{Miami}$,4,2,3, 3,$\underline{Miami}$,5,3,1) is not dominated by any other tuples and hence, it is a valid answer for the SQL query. Note that this query plan can be correct only when the join is a cartesian product.

Very recently, few research efforts have started to address preference queries over multiple relations [13], [14], [20], [27], however such work either focuses on only the skyline preference function [13], [14], [27] or provides a preliminary generic solution for a first cut generic preference query engine with no particular focus on the join operation [20]. Furthermore, two of these research efforts about skyline queries are mainly relying on the existence of index structures and are geared towards generating progressive results [14], [27].

In this paper, we propose *PrefJoin*; an efficient preference-aware join query operator. *PrefJoin* is generic for a wide variety of preference functions, does not assume the existence of any index structure, and achieves orders of magnitude performance over previous approaches [13], [20]. The main goal of *PrefJoin* is to make the join operation aware of the required preference functionality, and hence the join operation would be able to early prune those tuples that have no chance of being a preferred tuple without actually doing the join operation. The *PrefJoin* algorithm consists of four phases, namely, *Local Pruning*, *Data Preparation*, *Joining*, and *Refining*. The *Local Pruning* phase filters out, from each input relation those tuples that are guaranteed not to be in the final preference set. The *Data Preparation* phase associates meta data with each non-filtered tuple that will be used to optimize the execution of the next phases. The *Joining* phase uses that meta data, computed in the previous phase, to decide on which tuples should be joined together. Finally, the *Refining* phase finds the *final* preference set from the output of the joining phase.

PrefJoin is presented as a general framework that can support a wide variety of preference functions, though we only present three cases in this paper, namely, *skyline* [3], *multi-objective* [2], and *k-dominance* [5]. Experimental analysis of *PrefJoin*, implemented in PostgreSQL [26], shows from two to three orders of magnitude improvement over existing solutions [13], [20]. The rest of this paper is organized as follows: Section II formulates the problem. Section III highlights related research efforts. Section IV presents the *PrefJoin* framework with three case studies. Section V discusses the effect of the join order on the performance of the proposed algorithm. Section VI proves the correctness of the proposed algorithm. Section VII gives experimental analysis. Finally, Section VIII concludes the paper.

Algorithm	Query	Join Condition	Sorted Lists	Operator
TA & NRA [9]	Top-k Join	Key 1:1	√	—
KLEE [22]	Top-k Join	Key 1:1	√	—
J* [24]	Top-k join	General Equality	√	—
NRA-RJ [11]	Top-k join	Key 1:1	√	√
Rank-Join [12]	Top-k join	General Equality	√	√
Index [32]	Skyline join	Key 1:1	√	—
Multi-relational skyline [13]	Skyline Join	General Equality	—	√
Distributed Skyline Join [31]	Skyline Join	General Equality	—	—
FlexPref [20]	Preference Join	General Equality	—	√
PrefJoin	Preference Join	General Equality	—	√

TABLE I
RELATED WORK

II. Problem Formulation

Without loss of generality, we assume that all dimension values have a total order in which smaller values are better.

Problem Formulation. Given: (1) m input relations R_1, R_2, ..., R_m, (2) Equality join condition over R_1 to R_m, (3) A set of preference attributes $\mathcal{P}$; such that $\forall\ p \in \mathcal{P}$, $\exists\ i$ s.t $p \in R_i$ and $\forall\ R_i$, $R_i \cap \mathcal{P} \neq \phi$, and (4) A preference method $\mathcal{M}$. A preference query Q finds tuples from $R_1 \bowtie R_2 \bowtie \ldots R_m$, that are preferred with respect to $\mathcal{P}$ and $\mathcal{M}$.

Applying this formulation to the SQL query given in Section I gives: (1) Two input relations *Hotels H* and *Cruises C*, (2) The equality join condition is *H.location = C.location*, (3) Six preference attributes, *H.price*, *H.rating*, *H.beach_dist*, *C.price*, *C.rating*, and *C.days*, and (4) A skyline preference method. The SQL query finds those tuples, on the form (*HID*, *CID*, *Preference attributes*) that are skylines over the six preference attributes from $H \bowtie C$.

III. Related work

Due to their applicability, preference queries have received great interest ever since their introduction into databases. Several preference functions have been proposed in the literature including *skyline* [3], [8], [10], [16], [25], [34], *multi-objective* [2], *k-dominance* [5], *k-frequency* [6], *ranked skylines* [18], *spatial skyline* [29], *k representative skylines* [21], *distance-based dominance* [33], ϵ*-skyline* [35], and *top-k dominance* [37]. With the exception of very recent research [13], [14], [20], [27], all research efforts in preference query processing rely on the assumption that all input data reside in only one relation with no direct extension to support the case where preference attributes are scattered over more than one relation. Hence, the only solution is to completely join the input relations, then apply the preference method on the top of the join result, which is a very expensive solution.

Table I gives a taxonomy of existing work for preference queries over multiple relations. For completeness, we include a special case of join, where *one* relation is presented as a collection of sorted lists [9]. Each sorted list contains tuples on the form $(id, value)$ and is sorted based on the *value* attribute. Table I divides these research efforts with respect to: (a) Query type (e.g., Top-k join or skyline join), (b) Join condition (e.g., a general equality join condition or key only join), (c) Sorted lists (i.e., the input of the query must be in the form of sorted lists), and (d) Operator.

As it can be seen from table I, existing work for preference join can be divided into Top-k join, skyline join, and preference join. Furthermore, Top-k Join can be divided into: top-k over sorted lists (e.g., [9], [11]), approximate Top-k join in a distributed environment [22], and Top-k with general condition join [12], [24]. Other research efforts (e.g., [28], [30]), not shown in the table, compute the top-k join query over uncertain data. On the other hand, research efforts to compute skyline and preference join are limited to recent efforts [13], [14], [20], [27] (as seen from the table) which either focus on only the skyline preference function [13], [14], [27] or provide a preliminary generic solution for a general preference query engine with no particular focus on the join operation [20]. Furthermore, two of these research efforts mainly rely on the existence of index structures and are geared towards generating progressive results, turning them not optimized for a full join result [14], [27]. Other research efforts compute skyline join for a single relation, represented as set of sorted list as in [32], a centralized environment [13], [20] or in a distrusted environment [31]. Most of approaches designed for skyline and preference join (with except of [32]) support a generalized join condition.

There are two levels to embed the preference processing into database engine, as it can either implemented in application level as a layer in top of DBMS or as an operator inside the engine. When the operator is implemented inside the DBMS, the engine is well-aware of the existing of the operator and generates query plans that may use the operator. From the table, the algorithms that are presented as operators are [11]–[13]. Our proposed algorithms *PrefJoin* is implemented as an operator inside the PostgreSQL engine.

Our proposed approach *PrefJoin* distinguishes itself from its competitors [13], [14], [20], [27] as it has the following characteristics: (1) Unlike [13], [14], [27], *PrefJoin* is not limited to skyline queries, but it is generic to support a wide variety of preference queries; making it suitable for commercial database systems as it requires small code footprints for various preference functions, (2) Unlike [14], [27], *PrefJoin* does not assume the existence of any indexing data structure making nor geared towards producing progressive output, instead *PrefJoin* aims to support basic join queries that still lack the awareness of various preference functions, (3) On top of all other approaches, *PrefJoin* does not only employ elegant early punning techniques, but also, utilizes some meta information about input tuples to early decide whether two tuples should be joined together. With this, based on actual implementation inside PostgreSQL, *PrefJoin* achieves two to three orders of magnitude better performance over global skyline [13] for *skyline* queries and over FLexPref [20] for *skyline*, *k-dominance*, and *multi-objective* queries.

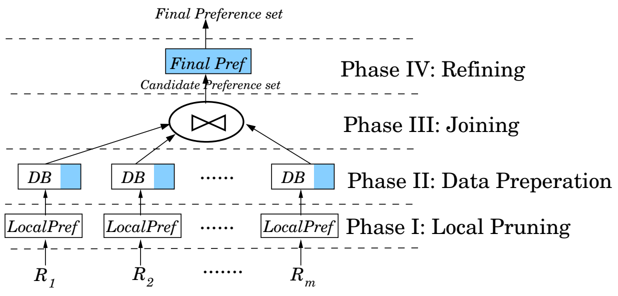


Fig. 2. Phases of *PrefJoin* Algorithm

IV. PrefJoin: A Preference-Aware Join Operator

In this section, we present our proposed algorithm, *PrefJoin*; an efficient preference-aware join query operator for a wide variety of preference functions. In particular, we focus, in this paper, on *skyline* [3], *k-dominance* [5] and *multi-objective* [2] preference functions. However, other preference functions that include *top-k dominating* [37] and *k-frequency* [6] can be also supported. The main goal of *PrefJoin* is to make the join operator aware of the required preference functionality, and hence the join operation would be able to prune those tuples that have no chance of being a preferred tuple with minimal computational overhead. The *PrefJoin* algorithm consists of four phases, as depicted in Figure 2, namely, *Local Pruning*, *Data Preparation*, *Joining*, and *Refining*. These phases use three preference functions $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ that are chosen carefully based on $\mathcal{P}$. Table II gives *possible* choices of $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for *skyline*, *k-dominance*, *multi-objective*, *top-k*, and *K-Frequency* preference functions. The choice of break-down for the input preference function $\mathcal{P}$ into $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ is arbitrary, and is the responsibility of the framework users to supply these functions. It is important to note that the break-down of a preference function $\mathcal{P}$ is not unique. For example, for any preference predicate $\mathcal{P}$, a naive and correct way of breaking $\mathcal{P}$ down is to set $\mathcal{P}_{local}$ and $\mathcal{P}_{pairwaise}$ to null, and $\mathcal{P}_{refine}$ to $\mathcal{P}$. The *Local Pruning* phase filters out, using $\mathcal{P}_{local}$, from each input relation, those tuples that are guaranteed not to be in the final preference set. The *Data Preparation* phase associates meta data with each non-filtered tuple that will be used to optimize the execution of the next phase. The *Joining* phase uses that meta data, computed in the previous phase, to decide on which tuples should be joined together. Finally, the *Refining* phase uses $\mathcal{P}_{refine}$ to find the *final* preference set from the output of the joining phase.

Sections IV-A - IV-D discuss the four phases of *PrefJoin* in a generic way that can support a wide variety of preference functions. Section IV-E gives the pseudo-code for the *PrefJoin* algorithm. Section IV-F gives three case studies of *PrefJoin*, namely, *skyline*, *k-dominance*, and *multi-objective* preference functions. Cost analysis for *PrefJoin* is presented in Appendix A. For simplicity, we limit our presentation of *PrefJoin* to the in-memory version. This is not a limitation for our contribution for two main reasons: (a) *PrefJoin* needs

	$\mathcal{P}_{local}$	$\mathcal{P}_{pairwise}$	$\mathcal{P}_{refine}$
Skyline [3]	*Skyline*	*Skyline*	*null*
K-dominance [5]	*Skyline*	*Skyline*	*K-dominance*
Multi-objective [2]	*Multi-Objective*	*Multi-Objective*	*Multi-objective*
Top-K [7]	*Top-k*	*Sorting*	*Rank-aware join*
K-Frequency [6]	*K-Frequency*	*null*	*K-Frequency*

TABLE II
POSSIBLE SETTING OF $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, AND $\mathcal{P}_{refine}$ FOR SOME PREFERENCES FUNCTIONS.

minimal memory requirements. Phase I needs only one in-memory hash bucket at a time, while Phases II and III can work with only two in-memory hash buckets at a time, and (b) memory spilling for hash-based joins is an independent research topic [19], [23]. The ideas for memory spilling can be orthogonally applied to *PrefJoin*, and its competitors (i.e., *GS* [13], and *FlexPref* [20]) without affecting their main operations nor performance trends.

A. Phase I: Local Pruning

Phase I filters out those tuples, from each input relation, that are guaranteed to be not in the final preference answer. The output of Phase I, i.e., the set of non-filtered tuples, is the local preference set $\mathcal{LP}(R_i)$ for each input relation R_i, which is defined as the set of tuples such that each tuple $t \in \mathcal{LP}(R_i)$ is a *preferred* tuple over all tuples in R_i with the same join attributes values. For example, Figure 1a highlights the tuples in the *local* preference set, for the *Hotels* and *Cruises* relations using the skyline preference method and the *location* attribute as the joining attribute. Phase I has the following two main steps: (a) *Hashing*, where Phase I scans each input relation R and utilizes a hash function h, based on the equality join attributes, to hash each tuple $t \in R_i$ to its corresponding hash bucket. For simplicity, we build a *separate* hash bucket B for each value of the equality join attributes. (b) *Local preference computation for each relation* R_i, where Phase I employs a preference function $\mathcal{P}_{local}$ over the set of tuples in each hash bucket $B \in R_i$ separately. It is important to note here that $\mathcal{P}_{local}$ does not have to be the same as $\mathcal{P}$, yet the choice of $\mathcal{P}_{local}$ depends on $\mathcal{P}$. For example, per Table II and as will be detailed in Section IV-F, if $\mathcal{P}$ is a *skyline*, *k-dominance*, or *multi-objective* preference functions, $\mathcal{P}_{local}$ would be *skyline*, *skyline*, or *multi-objective*, respectively.

The main idea of Phase I is that any tuple $t \in R_i$ that is not preferred, with respect to $\mathcal{P}_{local}$, over other tuples in R_i with the same value in the equality join attribute, should be filtered out as it has no chance of being preferred in $R_i \bowtie R_j$ with respect to $\mathcal{P}$. As t is not preferred to $\mathcal{P}_{local}$, there must be another tuple t' in the same hash bucket B of t that is better than t. This means that when joining R_i with relation R_j, the result of $t' \bowtie r_j$, $r_j \in R_j$, will be preferred over $t \bowtie r_j$. Such early pruning of t saves the overhead of considering t at later steps.

B. Phase II: Data Preparation

Phase II takes, as input, the local preference set, $\mathcal{LP}(R_i)$, for each relation R_i, produced from Phase I and passes it to Phase III along with a set of information, termed *Dominating hash buckets*, $DB(t)$, associated with each tuple $t \in \mathcal{LP}(R_i)$. Such information will be used later in Phase III to avoid producing unnecessary joined tuples. The main idea of Phase II is to associate with each local preference tuple t, produced from Phase I, the set of its dominating hash buckets, $DB(t)$, i.e., the set of hash buckets in R that contains tuples preferred over t with respect to preference function $\mathcal{P}_{pairwise}$. Same as $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$ does not have to be the same as $\mathcal{P}$, yet the choice of $\mathcal{P}_{pairwise}$ depends on $\mathcal{P}$. For example, per Table II and as will be detailed in Section IV-F, if $\mathcal{P}$ is a *skyline*, *k-dominance*, or *multi-objective*, $\mathcal{P}_{pairwise}$ would be *skyline*, *k-dominance*, or *skyline*, respectively.

For a local preference tuple t of bucket B in relation R, $DB(t)$ is computed by comparing t with the first tuple t' of each hash bucket B' in relation R, where $B' \neq B$. Three cases may occur:

- *Case 1: t' is preferred over t with respect to $\mathcal{P}_{pairwise}$.* In this case, we add bucket B' to $DB(t)$ as this means that there is a tuple in B' that is preferred over t. If $\mathcal{P}_{pairwise}$ is transitive, we guarantee that no other tuples in B can be preferred over t', thus no further preference checks are needed for other tuples in B'. On the other hand, if $\mathcal{P}_{pairwise}$ is not transitive, we proceed to compare t with the next tuple in B', and act accordingly based on our three cases.
- *Case 2: t is preferred over t' with respect to $\mathcal{P}_{pairwise}$.* In this case, we add bucket B to $DB(t')$ as this means that there is a tuple in B that is preferred over t'. If $\mathcal{P}_{pairwise}$ is transitive, we guarantee that no other tuples in B' can be preferred over t, thus no further preference checks are needed for other tuples in B'. On the other hand, if $\mathcal{P}_{pairwise}$ is not transitive, we proceed to compare t with the next tuple in B', and act accordingly based on our three cases.
- *Case 3: Neither t is preferred over t' nor t' is preferred over t.* In this case, we do not change neither $DB(t)$ nor $DB(t')$. We proceed with next tuple from B', and act accordingly based on our three cases.

C. Phase III: Joining

Phase III takes, as input, the local preference set $\mathcal{LP}(R_i)$ where each tuple $t \in \mathcal{LP}(R_i)$ is associated with a set of dominating hash buckets, $DB(t)$. Phase III uses $DB(t)$ to decide which local preference tuples $t_i \in \mathcal{LP}(R_i)$ and $t_j \in \mathcal{LP}(R_j)$ should be joined together, to produce the *candidate* preference set, denoted as $Candidate_{pref}$. This means that by only consulting the sets of dominating hash buckets, Phase III is able to decide if the joined tuple is a *candidate* preference tuple or not, rather than performing the join operation followed by a preference check. Basically, two tuples r and s from relations R and S that satisfy the equality join condition will be joined together only if $DB(r) \cap DB(s)=\phi$. Unlike all other phases, this phase does not directly depend on the preference function $\mathcal{P}$, as it always has the same execution regardless of $\mathcal{P}$.

For simplicity, we assume two input relations R and S, then we extend our ideas to support arbitrary number of input relations. Consider two local preference tuples r and s from corresponding hash bucket B of relations R and S, respectively. Two cases may arise:

- *Case 1: $DB(r) \cap DB(s) \neq \phi$, i.e., the sets of dominating hash buckets for r and s are overlapping.* In this case, we are sure that the joined tuple $t = r \bowtie s$ will not be preferred, hence, we avoid joining r and s. To illustrate, consider bucket $B' \in DB(r) \cap DB(s)$. There must be tuple $r' \in B'$, such that r' is preferred over r. Similarly, there must be $s' \in B'$, such that s' is preferred over s. This means that the tuple $t' = r' \bowtie s'$ must be preferred over t=$r \bowtie s$.
- *Case 2: $DB(r) \cap DB(s) = \phi$, i.e., the sets of dominating hash buckets for r and s are disjoint.* In this case, the joined tuple $t = r \bowtie s$ is a *candidate* preference tuple, i.e., it is a preferred tuple by separately considering attributes in relations R and S. Hence, we perform the join and produce tuple t.

The same idea is generalized to m input relations R_1, R_2 ,..., R_m, by joining the local preference tuples $t_1, t_2, \ldots, t_m$ from $R_1, R_2, \ldots, R_m$ only if $DB(t_1) \cap DB(t_2) \cap \ldots \cap DB(t_m) = \phi$. Hence, consulting the sets of dominating hash buckets is sufficient to check if the to be joined tuple is a *candidate* answer for the preference query.

D. Phase IV: Refining

Phase IV takes, as input, the *candidate* preference set, $Candidate_{pref}$, produced from the *joining* phase, and finds the final preference answer for the preference function $\mathcal{P}$. Simply, Phase IV employs a preference function $\mathcal{P}_{refine}$ over the set of tuples in the candidate preference set produced from Phase III. Each tuple t that is preferred with respect to $\mathcal{P}_{refine}$ is a final preference tuple for $\mathcal{P}$. It is important to note that $\mathcal{P}_{refine}$ does not have to be the same as $\mathcal{P}$, yet the choice of $\mathcal{P}_{refine}$ depends on $\mathcal{P}$. Specifically, $\mathcal{P}_{refine}$ is chosen to apply the preference computations that could not be pushed before the joining phase. For example, per Table II and as will be detailed in Section IV-F, if $\mathcal{P}$ is a *skyline*, *k-dominance*, or *multi-objective*, $\mathcal{P}_{refine}$ would be *null*, *k-dominance*, or *multi-objective*, respectively.

E. PrefJoin: Pseudocode

Algorithm 1 gives the pseudo code of the *PrefJoin* algorithm, presented for two relations R and S, for simplicity. The input to the algorithm is the two input relations R and S along with the three preference functions $\mathcal{P}_{local}$,$\mathcal{P}_{pairwise}$ and $\mathcal{P}_{refine}$ that are set based on the desired preference function $\mathcal{P}$, i.e., per Table II for *skyline*, *k-dominance*, and *multi-objective* preference functions. The algorithm starts by executing Phase I, i.e., building the hash buckets and computing the *local* preference sets $\mathcal{LP}(\mathcal{R})$ and $\mathcal{LP}(\mathcal{S})$ for the input relations R and S, respectively. This is achieved by applying the preference function $\mathcal{P}_{local}$ over each hash bucket in both input relations (Lines 3 to 10 in Algorithm 1). Then, we proceed to Phase II, where we compute the set of dominating hash buckets $DB(r)$ for each tuple $r \in \mathcal{LP}(\mathcal{R})$ and $DB(s)$ for each tuple $s \in \mathcal{LP}(\mathcal{S})$. $DB(r)$ and $DB(s)$ are computed as the set of hash buckets in R and S that include preferred tuple(s) over r and s, respectively, with respect to $\mathcal{P}_{pairwise}$ preference function (Lines from 12 to 17 in Algorithm 1). Then, we proceed to Phase III, as we initialize the candidate preference set, $Candidate_{pref}$ to be empty. We iterate over each pair of tuples (r,s) where r in hash bucket B in R and s in the corresponding hash bucket B in S. In this iteration, we compute $r \bowtie s$, and add its result to $Candidate_{pref}$ only if $DB(r) \cap DB(s) = \phi$ (Lines 19 to 24 in Algorithm 1). Finally, in Phase IV, if $\mathcal{P}_{refine}$ is null, we return $Candidate_{pref}$ as the final answer for preference function $\mathcal{P}$, otherwise, we apply the preference function $\mathcal{P}_{refine}$ over all entries in $Candidate_{pref}$ to produce the final answer for $\mathcal{P}$ (Lines 26 to 30 in Algorithm 1).

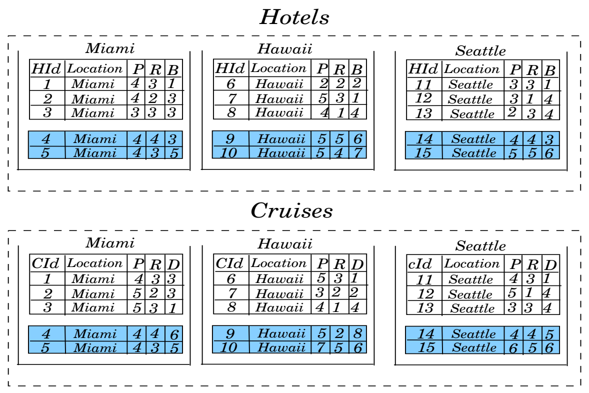
Algorithm 1 PrefJoin

```
1:  Function PrefJoin(Relation $R$, Relation $S$,$\mathcal{P}_{local}$,$\mathcal{P}_{pairwise}$,$\mathcal{P}_{refine}$)
2:  /* Phase I */
3:  $\mathcal{LP}(\mathcal{R}) \leftarrow \phi$;   $\mathcal{LP}(\mathcal{S}) \leftarrow \phi$
4:  Build hash buckets for relations $R$ and $S$
5:  for each Hash Bucket $B$ in relation $R$ do
6:      $\mathcal{LP}(\mathcal{R}) \leftarrow \mathcal{LP}(\mathcal{R}) \cup$ ApplyPreferenceFunction ($B$, $\mathcal{P}_{local}$)
7:  end for
8:  for each Hash Bucket $B$ in relation $S$ do
9:      $\mathcal{LP}(\mathcal{S}) \leftarrow \mathcal{LP}(\mathcal{S}) \cup$ ApplyPreferenceFunction ($B$, $\mathcal{P}_{local}$)
10: end for
11: /* Phase II*/
12: for each local preference tuple $r \in \mathcal{LP}(R)$ do
13:     $DB(r) \leftarrow$ The set of hash buckets in $R$ that include preferred tuple(s) over
        $r$ with respect to $\mathcal{P}_{pairwise}$ preference function
14: end for
15: for each local preference tuple $s \in \mathcal{LP}(S)$ do
16:     $DB(s) \leftarrow$ The set of hash buckets in $S$ that include preferred tuple(s) over
        $s$ with respect to $\mathcal{P}_{pairwise}$ preference function
17: end for
18: /* Phase III*/
19: $Candidate_{pref} \leftarrow \phi$
20: for each pair of tuples ($r$,$s$) where $r$ in hash bucket B in $R$ and $s$ in the
    corresponding hash bucket $B$ in $S$ do
21:     if ($DB(r) \cap DB(s) = \phi$) then
22:         Add $r \bowtie s$ to $Candidate_{pref}$
23:     end if
24: end for
25: /* Phase IV*/
26: if $\mathcal{P}_{refine}$ = Null then
27:     return $Candidate_{pref}$
28: else
29:     return ApplyPreferenceFunction ($Candidate_{pref}$,$\mathcal{P}_{refine}$)
30: end if
```

F. Case Studies

In this section, we show the strength of *PrefJoin* by giving three diverse case studies, namely, *skyline* [3], *multi-objective* [2] and *k-dominance skyline* [5]. Table II summarizes the $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for these preference functions. For space constraints, we use the SQL query presented in Section I, and relations *Hotels* and *Cruises* presented in Figure 1 as a running example. Generally speaking, we set $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ to $\mathcal{P}$.

1) Skyline: The *skyline* preference method returns tuples in a data set that are not dominated by (i.e., not strictly worse than) any other tuple in the data. Formally, given a dataset $\mathcal{D}$ of l-dimensional tuples, a *skyline* query finds each tuple



Hotels

Miami

HId	Location	P	R	B
1	Miami	4	3	1
2	Miami	4	2	3
3	Miami	3	3	3
4	Miami	4	4	3
5	Miami	4	3	5

Hawaii

HId	Location	P	R	B
6	Hawaii	2	2	2
7	Hawaii	5	3	1
8	Hawaii	4	1	4
9	Hawaii	5	5	6
10	Hawaii	5	4	7

Seattle

HId	Location	P	R	B
11	Seattle	3	3	1
12	Seattle	3	1	4
13	Seattle	2	3	4
14	Seattle	4	4	3
15	Seattle	5	5	6

Cruises

Miami

CId	Location	P	R	D
1	Miami	4	3	3
2	Miami	5	2	3
3	Miami	5	3	1
4	Miami	4	4	6
5	Miami	4	3	5

Hawaii

CId	Location	P	R	D
6	Hawaii	5	3	1
7	Hawaii	3	2	2
8	Hawaii	4	1	4
9	Hawaii	5	2	8
10	Hawaii	7	5	6

Seattle

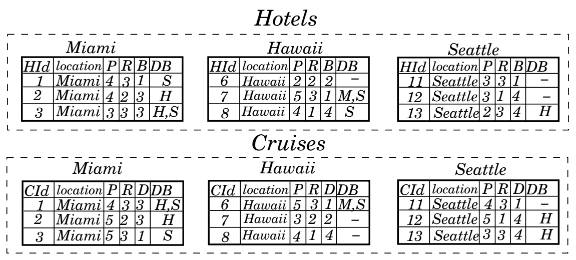
cId	Location	P	R	D
11	Seattle	4	3	1
12	Seattle	5	1	4
13	Seattle	3	3	4
14	Seattle	4	4	5
15	Seattle	6	5	6

Fig. 3. Phase I for *Skyline*, *multi-objective*, and *k-dominance* Queries

t, such that $\nexists\ t' \in \mathcal{D}$; s.t., $t'.p_i$ is better than or equal to $t.p_i$, for $1 \leq i \leq l$ and $t'.p_m$ is strictly better than $t.p_m$ for at least one dimension m. To support *skyline* queries within *PrefJoin*, we set $\mathcal{P}_{local}$ and $\mathcal{P}_{pairwise}$ to *Skyline*. Interestingly, we set $\mathcal{P}_{refine}$ to null, as each *candidate* preference tuple produced from Phase III is guaranteed to be a *final* preference function. This holds because the each tuple t_c=$r \bowtie s$ in the *candidate* preference set could not be dominated. Tuples that are preferred to tuple r, over preference attributes in R, are stored in hash buckets $DB(r)$. Similarly, those tuples that dominate s, over preference attributes in S, are only stored in the hash buckets $DB(s)$. Since $DB(r)$ and $DB(s)$ are disjoint, for each *candidate* preference tuple, it is impossible to find a single joined tuple that dominates t in both relations R and S. Thus, $r \bowtie s$ can not be dominated for *skyline* preference query, and it is a confirmed final answer. (A formal correctness proof is presented in Section VI).

Example. We apply *PrefJoin* algorithm for *skyline* preference function as follow:

Phase I. Figure 3 shows the hash buckets for the input relations, *Hotels* and *Cruises*, presented in Figure 1 where three hash buckets are built as one for each value of the location attribute, i.e., *Miami*, *Hawaii*, and *Seattle*. The set of discarded tuples are highlighted for each bucket in the two input relations. Other tuples represents the local preference $\mathcal{LP}(R)$, which is the output of Phase I. The hotel tuples h_9=(9,*$\underline{\text{Hawaii}}$*,5,5,6) and h_{10} = (10,*$\underline{\text{Hawaii}}$*,5,4,7) are locally dominated by hotel h_6=(6, *$\underline{\text{Hawaii}}$*,2,2,2), hence, they are not in the local preference set of Hawaii hash bucket. We can see that joining h_9 or h_{10} with any cruise in *Hawaii* (i.e., c_6 to c_{10}) will be dominated by joining h_6 with the same cruise. *Phase II.* Figure 4 gives the end result of Phase II after computing $DB(t)$ for all tuples in the *Hotels* and *Cruises* relations where M, H, and S refer to hash buckets *Miami*, *Hawaii*, and *Seattle*, respectively. To compute the set of dominating hash buckets for hotel h_1 = (1,*$\underline{M}$*,4,3,1), we compare h_1 with hotels in H and S hash buckets. For the first H hotel h_6 = (6,*$\underline{H}$*,2,2,2), neither h_1 dominates h_6 nor h_6 dominates h_1 (Case 3), so we do not change the set of dominating hash tuples for h_1 and h_6. Then, we proceed with h_7. We find that h_1 dominates



Hotels

Miami

HId	location	P	R	B	DB
1	Miami	4	3	1	S
2	Miami	4	2	3	H
3	Miami	3	3	3	H,S

Hawaii

HId	location	P	R	B	DB
6	Hawaii	2	2	2	–
7	Hawaii	5	3	1	M,S
8	Hawaii	4	1	4	S

Seattle

HId	location	P	R	B	DB
11	Seattle	3	3	1	–
12	Seattle	3	1	4	–
13	Seattle	2	3	4	H

Cruises

Miami

CId	location	P	R	D	DB
1	Miami	4	3	3	H,S
2	Miami	5	2	3	H
3	Miami	5	3	1	S

Hawaii

CId	location	P	R	D	DB
6	Hawaii	5	3	1	M,S
7	Hawaii	3	2	2	–
8	Hawaii	4	1	4	–

Seattle

CId	location	P	R	D	DB
11	Seattle	4	3	1	–
12	Seattle	5	1	4	H
13	Seattle	3	3	4	H

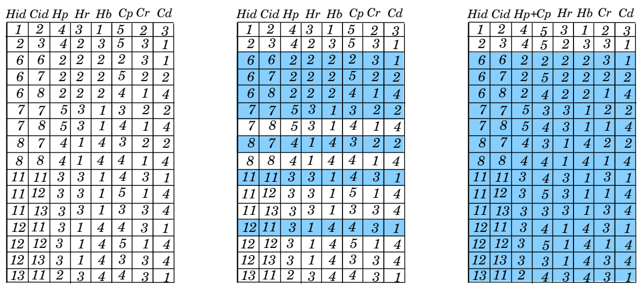
Fig. 4. Phase II for *Skyline*, *multi-objective*, and *k-dominance* Queries

tuple h_7 (Case 2). Hence, we add M to $DB(h_7)$. As no other hotel within H hash bucket can dominate h_1, we proceed with hotels from S hash bucket. Since h_{11} dominates h_1 (Case 1), we add S to $DB(h_1)$. Then, there is no need to continue checking other hotels in S as none of them can be dominated by h_1. *Phase III.* From Figure 4, the set of dominating hash buckets for h_1 is $DB(h_1)$={S}. Also for cruise c_1, we have $DB(c_1)$= {H,S}. Since $DB(h_1) \cap DB(c_1) \neq \phi$, there is no need to join h_1, c_1. Then, we proceed with the next cruise c_2 with $DB(C_2)$={H}. Since $DB(h_1) \cap SB(c_2)$ = ϕ, we do the actual join and add the results of Phase III. Figure 5a gives the *candidate* preference set.

Phase IV. No computations are needed in this phase, as each *candidate* preference tuple is guaranteed to be in the final answer.

2) K-dominance Skyline: A *k-dominance* preference query [5] redefines the traditional skyline dominance relation to consider only k dimensional subspaces, where k is less than or equal to the total number of preference attributes. Formally, given a dataset $\mathcal{D}$ of l-dimensional tuples, a k-dominance query finds each tuple t, such that $\nexists\ t' \in \mathcal{D}$; $t'.p_i$ is better than or equal to $t.p_i$ for at least k dimensions, and $t'.p_m$ is strictly better than $t.p_m$ for at least one dimension m.

Using *k-dominance* for $\mathcal{P}_{local}$ may discard tuples that are needed to eliminate non-preferred tuples in the final answer set, because *k-dominance* dominance relation is not transitive [5]. For example, consider tuples r_1=(1,$\underline{B}$,1,2,3), and r_2=(2,$\underline{B}$,2,1,4) in hash bucket B, and tuple r'=(1,$\underline{B'}$,3,1,2) in hash bucket B' in relation R, and tuples s=(1,$\underline{B}$,1,2,3) in hash bucket B, and s'=(1,$\underline{B'}$,1,2,3) in hash bucket B'. For 2-dominance skyline, r_1 2-dominates r_2, hence as described in Section IV-A, we remove tuple r_2 from $\mathcal{LP}(R)$. In Phase II, we calculate dominating hash buckets: $DB(r_1)$={B'}, $DB(r')$=ϕ, $DB(s)$=ϕ, $DB(s')$=ϕ. Hence, Phase III produces *candidate* preference set={$r_1 \bowtie s$, $r' \bowtie s'$}. In Phase IV, As $r' \bowtie s'$ 2-dominates $r_1 \bowtie s$, the *final* preference set is {$r_2 \bowtie s'$}. However, $r_2 \bowtie s$ 2-dominates $r' \bowtie s'$, therefore the correct *final* answer should be empty. Therefore, we could not use *k-dominance* for $\mathcal{P}_{local}$, and $\mathcal{P}_{pairwise}$. Thus, to be able to discard tuples in Phase I, we can use any transitive preference function f, such that for all possible input relation, the output of the transitive function f must be superset of the *k-dominance* preference. Therefore, we set P_{local} and $\mathcal{P}_{pairwise}$ to *skyline*. Setting $\mathcal{P}_{local}$ and



Hid	Cid	Hp	Hr	Hb	Cp	Cr	Cd
1	2	4	3	1	5	2	3
2	3	4	2	3	5	3	1
6	6	2	2	2	2	3	1
6	7	2	2	2	5	2	2
6	8	2	2	2	4	1	4
7	7	5	3	1	3	2	2
7	8	5	3	1	4	1	4
8	7	4	1	4	3	2	2
8	8	4	1	4	4	1	4
11	11	3	3	1	4	3	1
11	12	3	3	1	5	1	4
11	13	3	3	1	3	3	4
12	11	3	1	4	4	3	1
12	12	3	1	4	5	1	4
12	13	3	1	4	3	3	4
13	11	2	3	4	4	3	1

(a) Skyline

Hid	Cid	Hp	Hr	Hb	Cp	Cr	Cd
1	2	4	3	1	5	2	3
2	3	4	2	3	5	3	1
6	6	2	2	2	2	3	1
6	7	2	2	2	5	2	2
6	8	2	2	2	4	1	4
7	7	5	3	1	3	2	2
7	8	5	3	1	4	1	4
8	7	4	1	4	3	2	2
8	8	4	1	4	4	1	4
11	11	3	3	1	4	3	1
11	12	3	3	1	5	1	4
11	13	3	3	1	3	3	4
12	11	3	1	4	4	3	1
12	12	3	1	4	5	1	4
12	13	3	1	4	3	3	4
13	11	2	3	4	4	3	1

(b) K-dominance

Hid	Cid	Hp+Cp		Hr	Hb	Cr	Cd
1	2	4	5	3	1	2	3
2	3	4	5	2	3	3	1
6	6	2	2	2	2	3	1
6	7	2	5	2	2	2	2
6	8	2	4	2	2	1	4
7	7	5	3	3	1	2	2
7	8	5	4	3	1	1	4
8	7	4	3	1	4	2	2
8	8	4	4	1	4	1	4
11	11	3	4	3	1	3	1
11	12	3	5	3	1	1	4
11	13	3	3	3	1	3	4
12	11	3	4	1	4	3	1
12	12	3	5	1	4	1	4
12	13	3	3	1	4	3	4
13	11	2	4	3	4	3	1

(c) Multi-objective

Fig. 5. Example for Phase IV

$\mathcal{P}_{pairwise}$ to *skyline* would produce a superset of the answer set, therefore, to eliminate tuples that are dominated with respect to $k-dominance$, we set $\mathcal{P}_{refine}$ to *k-dominance*. (A formal correctness proof is presented in Section VI).

Example. We apply *PrefJoin* for *k-dominance* preference function as follow: *Phases I,II, and III*. As we are setting $\mathcal{P}_{local}$ and $\mathcal{P}_{pairwise}$ to *skyline*, exactly the same as *skyline* preference function, Phases I,II, and III proceed as presented earlier in Figure 3 and 4 for the *skyline*. *Phase IV.* The highlighted tuples in Figure 5b resembles the *final* preference set for *5-dominance* preference function. As an example, the candidate joined tuple $t_c = h_1 \bowtie c_2$ = (1,*Miami*,4,3,1,2, *Miami*,5,2,3) cannot be an answer for a five-dominance skyline query as it is 5-dominated by joined tuple $h_6 \bowtie c_7$ = (6,*Hawaii*,2,2,2,7,*Hawaii*,3,2,2).

3) Multi-objective: A *multi-objective* preference query [2] combines subsets of preference attributes using monotone scoring functions, and performs a skyline over the new transformed combined attributes. Formally, given a dataset $\mathcal{D}$ of l-dimensional tuples, and n monotone objective functions over tuple's attributes f_1, f_2, $\ldots$, f_n, a multi-objective query finds the set of tuples that are not dominated with respect to the objective functions. For our motivating example of Figure 1, a *multi-objective* query may sum the hotel price and cruise price into a single attribute and performs the skyline over five attributes: *total* price, hotel rating, hotel distance to beach, cruise rating, and cruise days.

To support *Multi-objective* queries within *PrefJoin*, we set the three preference functions: $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ to be *multi-objective* preference function. It is important to note that as we do not have all the input attributes to the objective functions in each input relation, while evaluating the objective functions, we substitute preference attributes from other input relations by a constant value (e.g., zero). (A formal correctness proof is presented in Section VI).

Example. We modify the SQL query given in Section I, to sum the hotel price and cruise price into a single attribute. Hence the *multi-objective* preference function have five attributes: *total* price, hotel rating, hotel distance to beach, cruise rating, and cruise days. We apply *PrefJoin* for *multi-objective* preference function as follow:

Phases I, II, and III. proceeds as *skyline* query because the given multi-objective function does not include any objective function that combines attributes from the same relation (Figure 3 and Figure 4). Figure 5b gives the *candidate* preference set, produced from the joining phase for *Hotels* and *Cruises* relations, depicted in Figure 1.

Phase IV. The highlighted tuples in Figure 5c resembles the *final* preference set for the given *multi-objective* preference function. As an example, the candidate joined tuple $t_c = h_1 \bowtie c_2$ = (1,*Miami*,4,3,1,2,*Miami*,5,2,3) cannot be an answer as it is dominated by joined tuple $h_6 \bowtie c_6$ = (6,*Hawaii*,2,2,2,6,*Hawaii*,2,3,1).

V. JOIN ORDER

The pseudo code of our *PrefJoin* algorithm, given in Section IV-E, highlights the following important observation. The set of dominating hashing buckets, $DB(s)$, is computed completely for each tuple $r\ in\ R$, where R is the outer input relation to the join operator. Then, for the inner input relation S, we only *partially* compute the set of dominating hash buckets, $DB(s)$, for each tuple $s \in S$. The main idea is that for each tuple $s \in S$, we compute only the dominating hash buckets *among* the ones in $DB(r)$, where r is the current tuple under consideration from relation R. This observation means that the overall performance of *PrefJoin* can be affected by the join order, i.e., having $R \bowtie S$ versus $S \bowtie R$. It is the objective of this section to estimate the cost of computing the sets of dominating hash buckets for both input relations R and S, should each relation be considered as an outer or an inner. Then, we use these costs to decide which join order will be more beneficial to the overall performance of *PrefJoin*. Using this analysis, we guarantee that our approach will always choose the right join order by selecting the inner relation to be the one with fewer local preference tuples.

Cost of computing $DB(R)$ for the outer relation R. For each tuple $r \in R$, where r is located in hash bucket B, the worst case cost of computing $DB(r)$ can be calculated by estimating the cost of comparing r pair wisely with each other local preference tuple in each other hash bucket B' from relation R, i.e., $B' \neq B$. Since the cost of comparing two tuples is proportional to the number of preference attributes, the total cost for computing $DB(r)$ is $Cost(DB(r))$= $\sum_B$ $(|B|*n)$ $\forall B$, s.t., $r \notin B$, where $|B|$ is the cardinality of hash bucket B in relation R, and n is the number of preference attributes in relation R. Summing up over all tuples in R, the total cost for local preference set computation for outer relation R is estimated to be $Cost_{DB}(R)$ = $\sum_r cost(DB(r))$, $\forall r \in \mathcal{LP}(R)$.

Cost of computing $DB(S)$ for the inner relation S. For each tuple $s \in S$, to be joined with tuple $r \in R$, where s is located in hash bucket B, the worst case cost of computing $DB(s)$ can be calculated by estimating the cost of comparing s pair wisely with each other local preference tuple *only* located in hash buckets $B \in DB(r)$. As the cardinality of $DB(r)$ is significantly lower than the number of hash buckets in S, having s as an inner relation encounters much lower cost in computing $DB(S)$, than having S as an outer relation. In our running example, the average cardinality of the sets of dominating hash buckets is 0.89, compared to 3 as the number of hash buckets. Then, the total cost for computing $DB(s)$ is $Cost(DB_r(s))$= $\sum_B$ $(|B|*m)$ $\forall B$, s.t., $B \in DB(r)$, where $|B|$ is the cardinality of hash bucket B in relation S, and m is the number of preference attributes in relation S. Summing up over all tuples in S, the total cost for local preference set computation for inner relation S is estimated to be $Cost_{DB}(S)$ = $\sum_s$ $cost(DB(s))$, $\forall s \in \mathcal{LP}(S)$.

Using the above cost estimations, the *PrefJoin* algorithm and pseudo code is slightly modified to perform the cost estimation procedure right away after Phase I. Basically, *PrefJoin* will contrast two costs: (a) The cost of having R as an outer relation plus the cost of having S as an inner relation, and (b) The cost of having S as an outer relation plus the cost of having r as an inner relation. If the first estimated cost is lower, we just proceed with R as an outer relation, otherwise, we swap the two relations to have S as the outer one.

VI. Proof of Correctness

This section proves the correctness of the *PrefJoin* algorithm for the *skyline*, *k-dominance* and *multi-objective* preference methods. For simplicity, we limit the correctness proof to two input relations R and S.

A. *Correctness of* PrefJoin *for* skyline *queries*

The correctness of *PrefJoin* for *skyline* preference function follows from proving that: (1) All skyline tuples in the joined relation $R \bowtie S$ are reported from the *PrefJoin* algorithm, and (2) Any tuple returned from the *PrefJoin* algorithm is a skyline over the joined relation $R \bowtie S$.

Theorem 1: Any tuple $r \bowtie s$ that is a skyline over the relation $R \bowtie S$, will be reported by the PrefJoin algorithm.

Proof: Assume that there exist a tuple t=$r \bowtie s$ that is a skyline over the relation $R \bowtie S$. However, t is not reported by the *PrefJoin* algorithm. Throughout the *PrefJoin* algorithm, if t is not reported, then this means that either tuple r or s (or both) was discarded in *Local Pruning* or *Joining* phases. In *Local Pruning* phase, a tuple r is only discarded if it is not a *local* preference tuple, i.e., there is a tuple $r' \in R$, and in the same hash bucket of r, such that r' dominates r. Since r and r' are in the same hash bucket, they have the same value for the join attribute, thus $r' \bowtie s$ dominates $r \bowtie s$, which contradicts our assumption that tuple $r \bowtie s$ is a skyline over the relation $R \bowtie S$. The same contradiction holds for s. In *Joining* phase, tuple $r \bowtie s$ is not added only if tuple s is dominated in the same bucket B where tuple r is dominated. Hence, there are other tuples $s' \in B'$, and $r' \in B'$. Thus, the joined tuple $r' \bowtie s'$ dominates $r \bowtie s$, which contradicts our assumption. We conclude that the assumption that t is not reported by *PrefJoin* is not possible. ■

Theorem 2: Any tuple $r \bowtie s$ that is reported by the *PrefJoin* algorithm is actually a skyline over the relation $R \bowtie S$.

Proof: Assume tuple t=$r \bowtie s$ is reported by the *PrefJoin* algorithm, but there is another tuple t'=$r' \bowtie s'$ that dominates t. Assume that $r' \in$ bucket B' of relation R and $s' \in$ bucket B' of relation S. Using the property of skyline dominance relation, r' must dominate r, and s' must dominate s. However, the algorithm adds a tuple to $Final_{sky}$, (Line 22 in Algorithm 1), only if s is not dominated in any bucket where r is dominated, including bucket B' which contains tuple s', as we assume that s' dominates s. Therefore, $r \bowtie s$ is not added to $Final_{sky}$, and hence not reported by *PrefJoin*, which contradicts our assumption. ■

B. *Correctness of* PrefJoin *for* k-dominance *queries*

The correctness of *PrefJoin* for *k-dominance* preference function follows from proving that: (1) All *k-dominance* tuples over the joined relation $R \bowtie S$ are reported from the *PrefJoin* algorithm, and (2) Any tuple returned from the *PrefJoin* algorithm is a *k-dominance* preference tuple over the joined relation $R \bowtie S$.

Theorem 3: All *k-dominance* tuples over the joined relation $R \bowtie S$ are reported from the *PrefJoin* algorithm.

Proof: As we set $\mathcal{P}_{local}$, and $\mathcal{P}_{pairwise}$ to *skyline*, from theorems 1 and 2, the *candidate* preference set contains all *skyline* tuples over the preference query. As the answer of *k-dominance* preference function is a subset of the answer of *skyline* preference function [5], *PrefJoin* returns all *k-dominance* preference tuple. ■

Theorem 4: Any tuple returned from the *PrefJoin* algorithm is a *k-dominance* preference tuple over the joined relation $R \bowtie S$.

Proof: As we set $\mathcal{P}_{refine}$ to *k-dominance* over the *candidate* preference set which contains all tuples in the answer of the *k-dominance* preference set (Theorem 3), each tuple returned from *PrefJoin* is a *k-dominance* preference tuple over the joined relation $R \bowtie S$. ■

C. *Correctness of* PrefJoin *for* multi-objective *queries*

The correctness of *PrefJoin* for *multi-objective* preference function follows from proving that: (1) All preference tuples with respect to *multi-objective* query in the joined relation $R \bowtie S$ are reported from the *PrefJoin* algorithm, and (2) Any tuple returned from the *PrefJoin* algorithm is a multi-objective preference tuple over the joined relation $R \bowtie S$.

Theorem 5: Any tuple t that is a preferred tuple with respect to *multi-objective* query over the relation $R \bowtie S$, will be reported by the *PrefJoin* algorithm.

Proof: Assume that there exists a tuple t that is a preferred tuple with respect to *multi-objective* query over the relation $R \bowtie S$. However, t is not reported by the *PrefJoin* algorithm. First assume that t=$r \bowtie s$ is a *preferred* tuple, yet it is not added to the *candidate* preference set. Throughout the *PrefJoin* algorithm, if t is not added to the *candidate* preference set, then this means that either tuple r or s (or both) was discarded in *Local Pruning* or *Joining* phases. In *Local Pruning* phase, a tuple r is only discarded if it is not a *local* preference tuple, i.e., there is a tuple $r' \in R$, and in the same hash bucket of r, such that r' dominates r over preference attributes in relation R, and for other attributes in S, we set them to constant values for tuples r and r' and objective functions are monotone. Since r and r' are in the same hash bucket, they have the same value for the join attribute, thus $r' \bowtie s$ dominates $r \bowtie s$, which contradicts our assumption that tuple $r \bowtie s$ is should be added to the *candidate* preference set. The same contradiction holds for s. In *Joining* phase, tuple $r \bowtie s$ is not added only if tuple s is dominated in the same bucket B where tuple r is dominated. Hence, there are other tuples $s' \in B'$, and $r' \in B'$. Thus, the joined tuple $r' \bowtie s'$ dominates $r \bowtie s$, which contradicts our assumption. This proves that t=$r \bowtie s$ is added to *candidate* preference set, if t is a preferred tuple. Then, as we set $\mathcal{P}_{refine}$ to *multi-objective*, then t must be reported if it is preferred with respect to *multi-objective* query. We conclude that the assumption that t is not reported by *PrefJoin* is not possible. ■

Theorem 6: Any tuple returned from the *PrefJoin* algorithm is a *multi-objective* preference tuple over the joined relation $R \bowtie S$.

Proof: As we set $\mathcal{P}_{refine}$ to *multi-objective* over the *candidate* preference set which contains all tuples in the answer of the *multi-objective* preference set (Theorem 5), each tuple returned from *PrefJoin* is a *multi-objective* preference tuple over the joined relation $R \bowtie S$. ■

VII. Experiments

In this section, we analyze the performance of our proposed framework, *PrefJoin*, compared with our two closest related works, the global skyline [13], denoted as *GS* and FlexPref [20], denoted as *Flex*. All our experiments are based on actual implementation of the three algorithms *PrefJoin*, *GS*, and *Flex*, inside PostgreSQL [26]. Unless mentioned otherwise, our data set is synthetically generated for two input relations R and S, where S is the inner relation, the ratio between the cardinality of both relations is 100, i.e., $\frac{|S|}{|R|}$=100, the cardinality of relation S is 1M, the number of groups (i.e., distinct values for the join attribute) is 5K, and the cardinality of the local preference set in each relation is 10% distributed uniformly over all the hash buckets. Also, we assume the set of preference attributes is distributed evenly between the two input relations, with a default of three attributes in each relation. We use the number of comparisons and wall clock time as our performance measures. We omit experiments over limited real data of hotels and restaurants extracted from Yelp website [36] due to space limitations and because it gives the same trends as the synthetic data. All experiments are executed on 2.0 Ghz Intel processor with 1 GB of RAM.

As we will see, *PrefJoin* always outperforms *GS* and *Flex* with at least one order of magnitude. That is why all the experiment figures depicted in this section are plotted with a log scale in terms of the number of comparisons or wall clock time.

A. Scalability

Figure 6 gives the behavior of the three algorithms, when increasing the cardinality of the inner relation S from 50K to 6.4M, while keeping the ratio between relations R and S intact. Figure 6a gives the number of comparisons in log scale for a skyline preference function, the order of magnitude difference between our proposed algorithm *PrefJoin* and other algorithms is due to the fact that *PrefJoin* avoids applying the preference function over the joined tuples, and utilizes the dominance relation between tuples in each relation. Similarly, Figures 6b and 6c give the number of comparisons for *k-dominance* and *multi-objective* respectively. As *GS* is only limited to *skyline*, we run our experiments using *Flex* and *PrefJoin*. The speedup for multi-objective query is smaller than *skyline*, and *k-dominance* preference function, as it requires more computations to be performed in Phase IV because objective preference attributes are not computed until Phase IV.

Figure 6d gives the wall clock time in log scale, for *skyline* preference query, which shows that *PrefJoin* has around two orders of magnitude better performance than other algorithms. The wall clock time for *multi-objective* and *k-dominance* shows similar behavior. Based on these experiments, we can conclude that, with the increase of the data size, *PrefJoin* is much more scalable than its competitors, and the performance gain reaches up to three orders of magnitude when the data size exceeds 1M. Due to space limitations, we run the next experiments using only the skyline preference function. This is also allow us to have *GS* in the experiments.

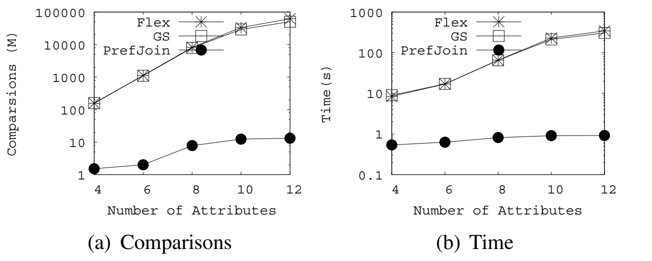


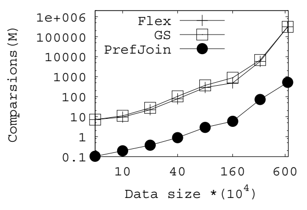
Fig. 7. Number of Preference Attributes

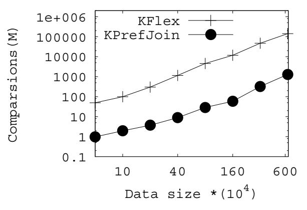
B. Number of Preference Attributes

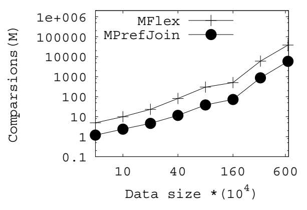
Figure 7 studies the effect of the number of preference attributes on the performance of *PrefJoin*, *GS*, and *Flex*, as we increase the number of preference attributes, in each relation, from two to six, i.e., increasing the total number of preference attributes of the output tuples from four to twelve. This directly increases the cardinality of the final preference set from 2K to 65K. For all algorithms, the number of comparisons and execution time increase with the increase of the number of preference attributes. However, *PrefJoin* exhibits better scalability as it avoids applying the preference function on the preference attributes of the joined tuples.

C. Join Cardinality

This section investigates the effect of the join cardinality on the execution time and number of comparisons. The join cardinality depends on: (a) the number of groups in each relation (i.e., the number of distinct values for the equality join attribute), and (b) the join ratio (i.e., how many tuples in S will be joined with a single tuple from R).


(a) Skyline


(b) K-dominance


(c) Multi-objective

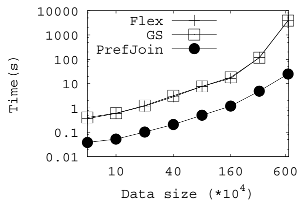
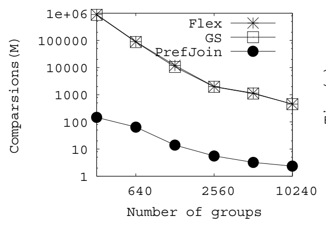

(d) Skyline

Fig. 6. Scalability


(a) Comparisons

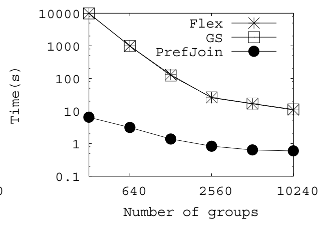
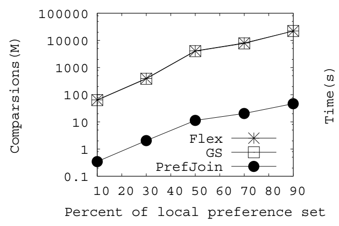

(b) Time

Fig. 8. Number of Groups


(a) Comparisons

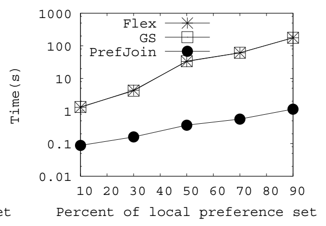
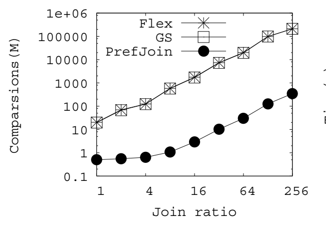

(b) Time

Fig. 10. Percentage of Local preference Sets


(a) Comparisons

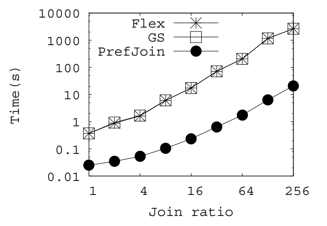
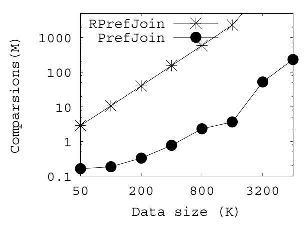

(b) Time

Fig. 9. Join ratio


(a) Comparisons

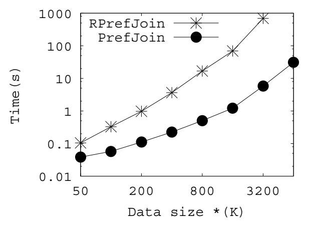

(b) Time

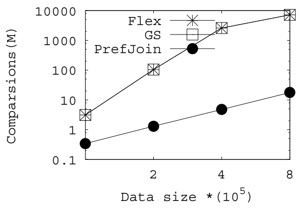
Fig. 11. Join Order

1) Number of Groups: Figure 8 studies the effect of increasing the number of groups (i.e., distinct values of the join attribute) from 300 to 10K for input relations, on the total runtime and the comparisons for *PrefJoin*, *Flex* and *GS* algorithms. With the increase of number of groups, the execution time and comparisons for all algorithms decrease exponentially where the size of the final preference set decreases from 260K to 5K. In the mean time, increasing the number of groups increases the cost of computing the sets of dominating hash buckets for local preference tuples, therefore the speedup of *PrefJoin* decreases with respect to *Flex* and *GS* algorithms. Overall, the *PrefJoin* algorithm exhibits at least two orders of magnitude better performance than both *GS* and *Flex* algorithms.

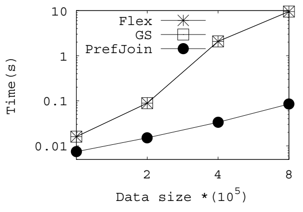
2) Join Ratio: Figure 9 increases the join ratio between relations R and S, i.e., how many tuples in S will be joined with a single tuple from R. The cardinality of R is set to 20K, while the join ratio is increased from 1:1 to 1:256. With the increase of the join ratio, execution time and comparisons, for all algorithms, increase as the size of the final preference set increases from 1320 to 111K. *PrefJoin*, consistently, has one to two orders of magnitude better performance than both *GS* and *Flex* algorithms, for all join ratios. This is due to: (a) The final preference set size increases; while *PrefJoin* avoids preference comparisons over these tuples, the other algorithms do not, and (b) utilizing the dominance relations in R to avoid unneeded dominance checks in S.

D. Percentage of Local Preference Set

Figure 10 gives the effect of increasing the percentage of the local preference set for relations R and S from 10% to 90%, on the total runtime and comparisons for *PrefJoin*, *Flex* and *GS*. We set the cardinality of relation S to 200K. Hence, the local preference set for relation S increases from 20K to 180K. With the increase of percentage of local preference set, the execution time and comparisons for all algorithms increase, yet *PrefJoin*, consistently, has at least two orders of magnitude better performance than both *GS* and *Flex* algorithms. This performance is due to the fact that as the final preference sets increase exponentially from 8K to 100K, only *PrefJoin* can utilize the dominance relations from relation R to avoid preference comparisons in S.



(a) Comparisons



(b) Time

Fig. 12. Three input relations

E. Join Order

Figure 11 studies the effect of the join order on the performance of the *PrefJoin* algorithm. We execute the *PrefJoin* algorithm on $R \bowtie S$, i.e., R is an outer relation and S is the inner one, and $S \bowtie R$, termed as *PrefJoin*. As most computations of the set of dominating hash buckets are executed in the outer relation of the join, *PrefJoin* shows an order of magnitude improvements for the running time and comparisons with the increase of the cardinality of relation S from 50K to 6.4M. This confirms the importance of the cost estimations discussed earlier in Section V. Due to accurate cost estimation, *PrefJoin* will decide to use R as the outer relation, and hence achieve an order of magnitude performance improvement in lieu of choosing S as the outer relation.

F. Multiple Input Relations

Figure 12 gives the performance of *PrefJoin*, *GS*, and *Flex* for three input relations when increasing the size of each relation from 100K to 800K. With the increase of input size, the number of comparisons and execution time increase for all algorithms. We can see that *PrefJoin* reaches up to four orders of magnitude better performance for the comparisons and three orders of magnitude better for the time than other algorithms. This is mainly because the *candidate* preference set increases exponentially with the input size for *GS* and *Flex* algorithms.

Contrasting this experiment with the similar one given in Figure 9 that were designed for only two relations, we can see that the performance gain achieved in *PrefJoin* over other algorithms is even better in case of three relations. This means that *PrefJoin* is better equipped with multiple input relations than other algorithms. This performance is due to two main factors: (1) The progressive output behavior of *PrefJoin* makes it easier to pipeline the result of one join to be the input of another join operator, such behavior is not found in other algorithms, and (2) Increasing the number of joined relations immediately results in increasing the number of preferred tuples which, as we have seen been seen in previous experiments, badly affected other algorithms.

We could not run other experiments for more than three input relations as the cost of executing both *GS* and *Flex* increases dramatically, and that shows us that *PrefJoin* is way preferable for multiple input relations.

VIII. Conclusion

This paper presented *PrefJoin*, an efficient preference-aware join query operator, designed specifically to deal with preference queries where the set of preferred attributes reside in more than one relation. The main idea of *PrefJoin* is to make the join operator aware of the required preference functionality, and hence inject the ability to early prune those tuples that have no chance of being a preferred tuple. *PrefJoin* consists of four main phases: *Local Pruning*, *Data Preparation*, *Joining*, and *Refining* that filter out, from each input relation, those tuples that are guaranteed not to be in the final preference set, associate meta data with each non-filtered tuple that will be used to optimize the execution of the next phases, produce a subset of join result that are relevant for the given preference function, and refine these tuples respectively. An interesting characteristic of *PrefJoin* is that it aims to join only those tuples that are guaranteed to be an answer, and hence: (a) saves computation costs by not joining unnecessary tuples, and (b) saves expensive preference computations by applying the preference function over those tuples that may needlessly be joined. *PrefJoin* supports a variety of preference function including skyline, multi-objective and k-dominance preference queries, by appropriately defining the $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for each preference function. The correctness of *PrefJoin* was proved as it returns all preferred tuples and all returned tuples are preferred. Experimental evaluation based on a system implementation of *PrefJoin* and its competitors [13], [20] inside PostgreSQL shows that *PrefJoin* consistently achieves one to three orders of magnitude performance gain over its competitors in various scenarios.

References

[1] M. J. Atallah and Y. Qi. Computing all skyline probabilities for uncertain data. In *Proceedings of the ACM Symposium on Principles of Database Systems, PODS*, 2009.

[2] W.-T. Balke and U. Güntzer. Multi-objective Query Processing for Database Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2004.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2001.

[4] C. Y. Chan, P.-K. Eng, and K.-L. Tan. Efficient Processing of Skyline Queries with Partially-Ordered Domains. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2005.

[5] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-Dominant Skylines in High Dimensional Space. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2006.

[6] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. On High Dimensional Skylines. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2006.

[7] S. Chaudhuri and L. Gravano. Evaluating Top-k Selection Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 1999.

[8] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2003.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *Proceedings of the ACM Symposium on Principles of Database Systems, PODS*, pages 102–113, Santa Barbara, CA, June 2001.

[10] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2005.

[11] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining Ranked Inputs in Practice. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2002.

[12] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting Top-k Join Queries in Relational Databases. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2003.

[13] W. Jin, M. Ester, Z. Hu, and J. Han. The Multi-Relational Skyline Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2007.

[14] W. Jin, M. D. Morse1, J. M. Patel, M. Ester, and Z. Hu. Evaluating Skylines in the Presence of Equijoins. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2010.

[15] W. Kießling. Foundations of Preferences in Database Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2002.

[16] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2002.

[17] G. Koutrika and Y. E. Ioannidis. Personalization of Queries in Database Systems. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2004.

[18] J. Lee, G. won You, and S. won Hwang. Personlized Top-K Skyline Queries in High-Dimensional Space. *Information Systems*, 34(1):45–61, 2009.

[19] J. J. Levandoski, M. E. Khalefa, and M. F. Mokbel. On producing high and early result throughput in multi-join query plans. In *TKDE*, 2010.

[20] J. J. Levandoski, M. F. Mokbel, and M. Khalefa. FlexPref: A Framework for Extensible Preference Evaluation in Database Systems. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2010.

[21] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting Stars: The k Most Representative Skyline Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2007.

[22] S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *VLDB*, 2005.

[23] M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, 2004.

[24] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting Incremental Join Queries on Ranked Inputs. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2001.

[25] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems, TODS*, 30(1):41–82, 2005.

[26] PostgreSQL: http://www.postgresql.org.

[27] V. Raghavan and E. A. Rundensteiner. Progressive result generation for multi-criteria decision support queries. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2010.

[28] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2007.

[29] M. Sharifzadeh and C. Shahabi. The Spatial Skyline Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2006.

[30] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k Query Processing in Uncertain Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2007.

[31] D. Sun, S. Wu, J. Li, and A. Tung. Skyline-join in distributed databases. In *ICDEW*, 2008.

[32] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2001.

[33] Y. Tao, L. Ding, X. Lin, and J. Pei. Distance-based Representative Skyline. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2009.

[34] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing Skyline Queries for Scalable Distribution. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2006.

[35] T. Xia, D. Zhang, and Y. Tao. On Skylining with Flexible Dominance Relation. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2008.

[36] Yelp: http://www.yelp.com.

[37] M. L. Yiu and N. Mamoulis. Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2007.

APPENDIX

A. *Cost Analysis of* PrefJoin

In this section, we compare the cost of computing preference queries for the proposed algorithm, *PrefJoin*, and *FlexPref* [20]. We limit our discussion only to *skyline* preference.

For two input relations R with cardinality of $|R|$ tuples and relation S with $|S|$ tuples. We denote the skyline tuples in relation R and S as $Sky(R)$, and $Sky(S)$, respectively. For simplicity, we assume a uniform distribution of skyline tuples in each input relation over k hash buckets, i.e., the cardinality of skyline tuples in hash bucket B of relation R is $\frac{|Sky(R)|}{k}$. However, our analysis is valid for arbitrary distribution. We compare the relevant performance cost of each phase for both *PrefJoin* and *FlexPref* algorithms, as follow:

- *Local Pruning*: *Local Pruning* phase finds the skyline for each input relations, this phase is identical in the both algorithms, hence it would not affect the relevant performance.
- *Data Preparation*: In *PrefJoin* algorithm, we find the dominance relation between the skyline tuples in local preference for each relation. This cost is bounded by $O(|Sky(R)|^2)$ and $O(|Sky(S)|^2)$ for relation R and S, respectively. Hence, the total cost of *Data Preparation* is $O(|Sky(R)|^2 + |Sky(S)|^2)$. However, *FlexPref* does not performs any data preparation.
- *Joining*: The cost of joining local preference set for hash bucket B from relation R with the corresponding local preference set in relation S, for both approaches, is bounded by $O(\frac{|Sky(R)|}{k}.\frac{|Sky(S)|}{k})$, as the number of tuples is $\frac{|Sky(R)|}{k}$, and $\frac{|Sky(S)|}{k}$ respectively. Hence, the total join cost is bounded by $O(\frac{|Sky(R)|.|Sky(S)|}{k})$ join operations. The cardinality of the join result is bounded by $O(\frac{|Sky(R)|.|Sky(S)|}{k})$ tuples.
- *Refining*: *Refining* phase, in *PrefJoin*, is not needed for skyline queries, i.e., $O(1)$, while, the cost to find the skyline over the joined tuples is bounded by $O(\frac{|Sky(R)|.|Sky(S)|}{k}^2)$ comparisons.

From this analysis, we can deduce that our approach performs $O(|Sky(R)|^2 + |Sky(S)|^2)$ comparisons in *Data Preparation* phase to save $O(\frac{|Sky(R)|.|Sky(S)|}{k}^2)$ comparisons.