PrefJoin: An Efficient Preference-aware Join Operator

Mohamed E. Khalefa  Mohamed F. Mokbel  Justin J. Levandoski

Department of Computer Science and Engineering, University of Minnesota
{khaleda,mokbel,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 discard irrelevant tuples from the input relations, prepare tuples for next phases, joins non-pruned objects, and refine the join result respectively. PrefJoin supports a variety of preference function including skyline, multi-objective and k-dominance preference queries. An interesting characteristic of PrefJoin is that it is tightly integrating with join hence we can early prune join only tuples that are guaranteed not to be an answer, and hence it saves significant unnecessary computations cost. 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 objects, preference queries find a set of interesting objects, i.e., objects 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], [23], [32]). 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. Consider, for example, a scenario where a user is looking for a hotel and a cruise in the same location. 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:

```sql
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 ⋈ 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 objects. 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 ⋈ location Cruises) ≠ Pref(Hotels) ⋈ location Pref(Cruises). For example, in Figure 1, if the preference function is a skyline query, the hotel represented by tuple (2.Miami,4.2,3) is dominated by hotel (6.Hawaii,2.2,2), hence it would not proceed to
the join operator. Similarly, cruise \((3, \text{Miami}, 5, 3, 1)\) is dominated by cruise \((11, \text{Seattle}, 4, 3, 1)\). However, the joined tuple \((2, \text{Miami}, 4, 2, 3, 3, \text{Miami}, 5, 3, 1)\) 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], [19], [25], however such work either focuses on only the skyline preference function [13], [14], [25] or provides a preliminary generic solution for a first cut generic preference query engine with no particular focus on the join operation [19]. 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], [25].

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], [19]. 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 prune those tuples that have no chance of being a preferred object 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 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 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 the PrefJoin algorithm, implemented in PostgreSQL [24], shows from two to three orders of magnitude improvement over existing solutions [13], [19]. 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.

## 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 \(R_1, R_2, \ldots, R_m\), (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 \(\forall R_i, R_j \cap \mathcal{P} \neq \emptyset\), and (4) A preference method \(\mathcal{M}\). A preference query \(Q\) finds tuples from \(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m\), 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 \(\text{Hotels H and Cruises C}\), (2) The equality join condition is \(\text{H.location} = \text{C.location}\), (3) Six preference attributes, \(\text{H.price, H.rating, H.beach_dist, C.price, C.rating, and C.days}\), and (4) A skyline preference method. The SQL query finds those tuples, on the form \((\text{HID, CID, Preference attributes})\) that are skylines over the six preference attributes from \(\text{H} \bowtie \text{C}\).

### 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], [23], [32], multi-objective [2], k-dominance [5], k-frequency [6], ranked skylines [18], spatial skyline [27], k representative skylines [20], distance-based dominance [31], \(\epsilon\)-skyline [33], and top-k dominance [34]. With the exception of very recent research [13], [14], [19], [25], 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 consider 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 \((\text{id, value})\) 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 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 in the table, existing work for preference join are Top-K Join for one relation represented as a collection

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Query Type</th>
<th>Join Condition</th>
<th>Sorted Lists</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA &amp; NRA [9]</td>
<td>Top-k Join</td>
<td>Key 1:1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KLEE [21]</td>
<td>Top-k Join</td>
<td>Key 1:1</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>J* [22]</td>
<td>Top-k join</td>
<td>General</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rank-Join [12]</td>
<td>Top-k Join</td>
<td>General</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index [10]</td>
<td>Skyline Join</td>
<td>Key 1:1</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Distributed Skyline Join [29]</td>
<td>Preference Join</td>
<td>General</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FlexPref [19]</td>
<td>Preference Join</td>
<td>General</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

**TABLE I**

**Related work**

Given: (1) Two input relations \(\text{Hotels H and Cruises C}\), (2) The equality join condition is \(\text{H.location} = \text{C.location}\), (3) Six preference attributes, \(\text{H.price, H.rating, H.beach_dist, C.price, C.rating, and C.days}\), and (4) A skyline preference method. The SQL query finds those tuples, on the form \((\text{HID, CID, Preference attributes})\) that are skylines over the six preference attributes from \(\text{H} \bowtie \text{C}\).
of sorted list (e.g., [9], [11]), approximate Top-k join in a distributed environment [21], Top-K with general condition join [12], [22], skyline Join for one relation represented in sorted manner [30], skyline Join in a centralized environment [13], [19], skyline Join in a distrusted environment [29], preference join [19]. Other research efforts [26], [28], not shown in the table, compute the Top-k join query over uncertain data.

Existing work for multi-preference query processing (i.e., no top-k) is limited to the very recent research [13], [14], [19], [25] (as seen from the table) which either focus on only the skyline preference function [13], [14], [25] or provide a preliminary generic solution for a general preference query engine with no particular focus on the join operation [19]. 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], [25]. The two closest works to ours are: (a) global skyline [13], as it does not require an index structure and not geared towards progressive results, though it supports only skyline queries, and (b) FlexPref [19], as it is kind of a generic solution for a wide variety of preference functions. Both approaches [13], [19] have similar skeleton functionality that contains three steps: (1) Computing the local preference set for each input relation $R_i$, (2) Joining local preference tuples from input relations to find candidate preference set, and (3) Computing the preference function over the candidate preference set.

There are two levels to embed the preference queries into database either in application level as a layer in top of DBMS or inside the engine. When the operator is implemented inside the DBMS, the engine is well-aware of the existing of the operator and generates plans that uses 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 DBMS engine.

There are two levels to embed the preference queries into database either in application level as a layer in top of DBMS or inside the engine. When the operator is implemented inside the DBMS, the engine is well-aware of the existing of the operator and generates plans that uses 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 DBMS engine.

Our proposed approach PrefJoin distinguishes itself from its competitors [13], [14], [19], [25] as it has the following characteristics: (1) Unlike [13], [14], [25], 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], [25], 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 pruning 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 [19] for skyline, $k$-dominance, and multi-objective queries.

### 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 [34] 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 object 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 $P_{local}$, $P_{pairwise}$, and $P_{refine}$ that are chosen carefully based on $P$. Table II gives the choices of $P_{local}$, $P_{pairwise}$, and $P_{refine}$ for skyline, $k$-dominance and multi-objective preference functions. The Local Pruning phase filters out, using $P_{local}$, 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 $P_{refine}$ 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.
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 \( \mathcal{P}_{\text{local}} \) over the set of tuples in each hash bucket \( B \in R_i \) separately. It is important to note here that \( \mathcal{P}_{\text{local}} \) does not have to be the same as \( \mathcal{P} \), yet the choice of \( \mathcal{P}_{\text{local}} \) 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, \( \mathcal{P}_{\text{pairwise}} \) 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 \( \mathcal{P}_{\text{pairwise}} \). 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 \( \mathcal{P}_{\text{pairwise}} \) 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 side, if \( \mathcal{P}_{\text{pairwise}} \) 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 \( \mathcal{P}_{\text{pairwise}} \). 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 \( \mathcal{P}_{\text{pairwise}} \) 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 side, if \( \mathcal{P}_{\text{pairwise}} \) 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 \( \text{Candidate}_{\text{pref}} \). 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 \( DB(r) \cap DB(s) \neq \phi \). 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 \( t = r \sqcap s \) will not be
The input to the algorithm is the two input relations along with the three preference functions. The algorithm starts by building hash buckets for relations $R$ and $S$. For each hash bucket $B$ in relation $R$, a set of tuples is a final preference tuple for $P$ as the set of hash buckets in $R$ that include preferred tuple(s) over $r$ with respect to $P_{\text{pairwise}}$ preference function.

### D. Phase IV: Refining

Phase IV takes as input the candidate preference set, $\text{Candidate}_{\text{pref}}$, produced from the joining phase, and finds the final preference answer for the preference function $P$. Simply, Phase IV employs a preference function $P_{\text{refine}}$ over the set of tuples in the candidate preference set produced from Phase III. Each tuple $t$ that is preferred with respect to $P_{\text{refine}}$ is a final preference tuple for $P$. It is important to note that $P_{\text{refine}}$ does not have to be the same as $P$, yet the choice of $P_{\text{refine}}$ depends on $P$. Specifically, $P_{\text{refine}}$ 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 $P$ is a skyline, $k$-dominance, or multi-objective, $P_{\text{refine}}$ 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 $P_{\text{local}}$, $P_{\text{pairwise}}$, and $P_{\text{refine}}$ that are set based on the desired preference function $P$, 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 $LP(R)$ and $LP(S)$ for the input relations $R$ and $S$, respectively. This is achieved by applying the preference function $P_{\text{local}}$ 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 $LP(R)$ and $DB(s)$ for each tuple $s$ in $LP(S)$. $DB(r)$ 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 $P_{\text{pairwise}}$ preference function (Lines from 12 to 17 in Algorithm 1). Then, we proceed to Phase III, as we initialize the candidate preference set, $\text{Candidate}_{\text{pref}}$, 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 $(r, s)$, and add its result to $\text{Candidate}_{\text{pref}}$ only if $DB(r) \cap DB(s) = \phi$ (Lines 19 to 24 in Algorithm 1). Finally, in Phase IV, if $\text{Candidate}_{\text{pref}}$ is null, we return $\text{Candidate}_{\text{pref}}$ as the final answer for preference function $P$, otherwise, we apply the preference function $P_{\text{refine}}$ over all entries in $\text{Candidate}_{\text{pref}}$ to produce the final answer for $P$ (Lines 26 to 30 in Algorithm 1).

### 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 $P_{\text{local}}$, $P_{\text{pairwise}}$, and $P_{\text{refine}}$ for these preference functions. 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 $P_{\text{local}}$, $P_{\text{pairwise}}$, and $P_{\text{refine}}$ to $P$.

1) Skyline: The skyline preference method returns objects in a data set that are not dominated by (i.e., not strictly worse than) any other object in the data. Formally, given a dataset $D$ of $l$-dimensional tuples, a skyline query finds each tuple $t$, such that $\nexists t' \in D : s.t., t.p_i$ 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 $P_{\text{local}}$ and $P_{\text{pairwise}}$ to Skyline. Interestingly, we set $P_{\text{refine}}$ 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 $t=r \bowtie s$ in the candidate preference set could not be dominated. Tuples that are 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 $t=r \bowtie s$.

2) 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 $R_1, R_2, \ldots, R_m$, 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.
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, \( r \not\preceq s \) 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 \( h_0=(9, \text{Miami}, 5.5, 6) \) and \( h_10=(10, \text{Hawaii}, 5.4, 7) \) are locally dominated by hotel \( h_0=(6, \text{Hawaii}, 2.2, 2) \), hence, they are not in the local preference set of Miami hash bucket. We can see that joining \( h_0 \) or \( h_10 \) with any cruise in Hawaii (i.e., \( c_0 \) to \( c_10 \)) will be dominated by joining \( h_0 \) 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 \( h_1 = (1, \text{Miami}, 4.3, 1) \), we compare \( h_1 \) with hotels in \( H \) and \( S \) hash buckets. For the first \( H \) hotel \( h_0 = (6, \text{Hawaii}, 2.2, 2) \), neither \( h_1 \) dominates \( h_0 \) nor \( h_0 \) dominates \( h_1 \) (Case 3), so we do not change the set of dominating hash tuples for \( h_1 = (1, \text{Hawaii}, 4.3, 1) \) and \( h_0 \). Then, we proceed with \( h_7 \). We find that \( h_1 \) dominates 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 \( h_1 \) 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 \( DB(h_1) \cap DB(c_1) \neq \emptyset \), 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 \( DB(h_1) \cap DB(c_2) = \emptyset \), 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 \( D \) of \( l \)-dimensional tuples, a k-dominance query finds each tuple \( t \), such that \( \exists t' \in D; t' \preceq_p t \) 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 \( r_1 = (1, B, 1, 2, 3) \), and \( r_2 = (2, B, 2, 1, 4) \) in hash bucket \( B \), and tuple \( r' = (1, B', 3, 1, 2) \) in hash bucket \( B' \) in relation \( R \), and tuples \( s = (1, B, 1, 2, 3) \) in hash bucket \( B \), and \( s' = (1, B', 1, 2, 3) \) 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: \( DB(r_1) = \{B'\} \), \( DB(r') = \emptyset \), \( DB(s) = \emptyset \), \( DB(s') = \emptyset \). Hence, Phase III produces candidate preference set \( \{r_1 \notin s, r' \not\preceq s'\} \). In Phase IV, As \( r' \not\preceq s' \) 2-dominates \( r_1 \not\preceq s \), the final preference set is \( \{r_2 \not\preceq s'\} \). However, \( r_2 \not\preceq s \) 2-dominates \( r' \not\preceq s' \), 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 \( \mathcal{P}_{local} \) and \( \mathcal{P}_{pairwise} \) to skyline. Setting \( \mathcal{P}_{local} \) and \( \mathcal{P}_{pairwise} \) to skyline would produce a superset of the answer set, therefore, to eliminate tuples that are dominated with respect to \( k \) dominance, we set \( \mathcal{P}_{Prefine} \) to k-dominance. (A formal correctness proof is presented in Section VI).

Example. We apply PrefJoin for k-dominance preference function as follow: Phases II, 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 $t_c = h_1 \bowtie c_2 = (1, \text{Miami}, \{4, 3, 1, 2\}, \text{Miami}, \{5, 2, 3\})$ cannot be an answer for a five-dominance skyline query as it is 5-dominated by joined tuple $h_6 \bowtie c_7 = (6, \text{Hawaii}, \{2, 2, 2, 6\}, \text{Hawaii}, \{2, 3, 1\})$.

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 $D$ 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: $P_{local}$, $P_{pairwise}$, and $P_{refine}$ 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 $t_c = h_1 \bowtie c_2 = (1, \text{Miami}, \{4, 3, 1, 2\}, \text{Miami}, \{5, 2, 3\})$ cannot be an answer as it is dominated by joined tuple $h_6 \bowtie c_7 = (6, \text{Hawaii}, \{2, 2, 2, 6\}, \text{Hawaii}, \{2, 3, 1\})$.

V. JOIN ORDER

The pseudo code of our PrefJoin algorithm, given in Section IV-E, highlights a very important observation, stemmed out from the fact that of reducing the cost of computing the preference over the fewer joined results than other approaches. Basically, 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.

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 $\text{Cost}(DB(r)) = \sum_B (|B|*n) \forall B, r \notin B$, 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 $\text{Cost}_{DB}(R) = \sum_r \text{Cost}(DB(r))$, $\forall r \in \mathcal{L}(R)$.

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 $DB(r)$ 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 $\text{Cost}(DB_s(s)) = \sum_B (|B|*m) \forall B, s \in DB(r)$, 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 $\text{Cost}_{DB}(S) = \sum_s \text{cost}(DB(s)), \forall s \in \mathcal{L}(P(S))$.

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 $r'$ does not dominates $r$. Since $r$ and $r'$ are in the same hash bucket, they have the same value for the join attribute, thus $r'$ does not dominates $s$, 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 $r' \in B'$ of relation $R$ and $s' \in B'$ 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 $\text{Final}_{sk}$. (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 $\text{Final}_{sk}$, 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 $r$. Since $r$ and $r'$ are in the same hash bucket, they have the same value for the join attribute, thus $r'$ does not dominates $r$. Since $r$ and $r'$ are in the same hash bucket, they have the same value for the join attribute, thus $r'$ does not dominates $s$, which contradicts our assumption. We conclude that the assumption that $t$ is not reported by PrefJoin is not possible.
$B$ 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 $r \bowtie s$ is added to candidate preference set, if $t$ is a preferred tuple. Then, as we set $\mathcal{P}_{\text{refine}}$ 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 $\mathcal{P}_{\text{refine}}$ 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 [19], denoted as Flex. All our experiments are based on actual implementation of the three algorithms PrefJoin, GS, and Flex, inside PostgreSQL [24]. 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%. 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. 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.

### 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 $R$).

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.

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 ▷◁ S, i.e., R is an outer relation and S is the inner one, and S ▷◁ R, termed as RPrefJoin. 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 object. PrefJoin consists of four main phases: Local Pruning, Data Preparation, Joining, and Refining that discard irrelevant tuple from the input relations, prepare tuples for next phases, joins non-pruned objects, and refine the join result 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 objects, and (b) saves computation cost in pruning, by applying the preference function over those objects 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 \( P_{\text{local}} \), \( P_{\text{pairwise}} \), and \( P_{\text{refine}} \) 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], [19] inside PostgreSQL show that PrefJoin consistently achieves one to three orders of magnitude performance gain over its competitors in various scenarios.

### References


APPENDIX

A. Cost Analysis of PrefJoin

In this section, we compare the cost of computing preference queries for the proposed algorithm, PrefJoin, and FlexPref [19]. 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 \( \frac{|Sky(R)|}{k} \).

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 \( O\left(\frac{|Sky(R)|}{k} \cdot \frac{|Sky(S)|}{k}\right) \), as the number of tuples is \( \frac{|Sky(R)|}{k} \) and \( \frac{|Sky(S)|}{k} \) respectively. Hence, the total join cost is bounded by \( O\left(\frac{|Sky(R)|}{k} \cdot |Sky(S)|^2\right) \) 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 \( O\left(\frac{|Sky(R)|}{k} \cdot |Sky(S)|^2\right) \) comparisons.

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